Theoretical Computer Science 22 (1983) 297-315
North-Holland Publishing Company

OPTIMAL OFF-LINE DETECTION OF REPETITIONS IN A STRING*

A. APOSTOLICO** and F.P. PREPARATA
Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL 61801, U.S.A.

Communicated by M. Nivat
Received June 1981
Revised March 1982

Abstract. An algorithm is presented to detect—within optimal time $O(n \log n)$ and space $O(n)$, off-line on a RAM—all of the distinct repetitions in a given textstring on a finite alphabet. The proposed strategy is self-contained, as it depends more heavily on algorithmic design considerations than on the combinatorial properties of the output It is based on a new data structure, the leaf-tree, which is particularly suited to exploit simple properties of the suffix tree associated with the string to be analyzed.

1. Introduction

Strings of symbols containing no consecutive occurrences of the same pattern have attracted the attention of researchers in diverse fields for a long time. Perhaps their first appearance dates back to the work by Thue [16], who is generally credited with the discovery of arbitrarily long streams of symbols from a finite alphabet that do not contain any 'square' substrings, i.e., subpatterns formed by the concatenation of some substring with itself.

In recent years, the study of such 'square-free' strings has been found relevant to automata and formal language theory, algebraic coding and more generally in systems theory and combinatorics, and we shall make no attempt to refer to the existing copious literature. Suffice it to mention that papers have been devoted to the construction of arbitrarily long square-free (as well as other related repetition-constrained) strings [3, 7, 8] over alphabets of fixed cardinality. In a related endeavor, the complementary notions of periodicity and overlaps of strings have been extensively investigated and still are an active research subject (see Duval [5] for an extensive bibliography).

* This work was partially supported by the National Science Foundation Grant MCS-78-13642, by the Joint Services Electronics Program Contract N00014-79-C-0424, and by a Grant by G.N.I.M. (Gruppo Nazionale di Informatica Matematica), Consiglio Nazionale delle Ricerche, Italy. Additional support was provided by NATO Research Grant 039.82.
** On leave from the Istituto di Scienze dell' Informazione, Università di Salerno, Italy.

In the framework of pattern matching [1], some classic results on string periodicity [6, 12] have been used to develop clever techniques for the detection of assigned patterns in textstrings in time linear in the string length [10].

The problem of the efficient recognition of the occurrence of substring squares in a string stems quite naturally from the preceding remarks, and it is certainly relevant to a variety of practical applications as well [2]. $O(n^2)$-time algorithms can be quickly developed on the basis of existing pattern matching techniques and tools. Recently, an $O(n \log n)$ algorithm has been proposed to determine whether a given textstring over a finite alphabet contains a repetition [13]. During the preparation of this paper, M. Crochemore [4] developed an $O(n \log n)$ algorithm to determine all repetitions in a textstring x. Crochemore's approach essentially relies on the well-known minimization algorithm for finite state automata [1] and exploits the theoretical bound of $|x| \log |x|$ repetitions in a string [11] as a terminating condition ($|x|$ denotes the length of x).

In this paper, we present a more direct algorithmic criterion for the latter problem. The proposed strategy basically relies on the properties of suffix trees [14] associated with textstrings, but makes crucial use of a novel structure, called leaf-tree. The resulting algorithm is still inherently off-line and takes $O(n \log n)$ time and $O(n)$ space in the worst case.

2. Preliminaries

Let I be a finite alphabet and I^+ the free semigroup generated by I. A *string* $x \in I^+$ is fully specified by writing $x = a_0a_1 \cdots a_{n-1}$, where $a_i \in I$ $(i = 0, 1, \ldots, n-1)$ and $|x|$ denotes the length of x. We assume here that x is stored as an array $x[0:n-1]$, where $x[i] = a_i$ $(i = 0, 1, \ldots, n-1)$. Given $x = a_0a_1 \cdots a_{n-1}$, w is a *substring* of x if there exist indices i, j $(0 \leq i \leq j \leq n-1)$ such that $w = a_ia_{i+1} \cdots a_j$. A *factor* of x is a substring of x and its starting index in $\{0, 1, \ldots, n-1\}$ (that is, a positioned substring). The notation $x[i:j]$ is used to denote the factor of $x : x[i]x[i+1] \cdots x[j]$. A left (right) factor of x is a *prefix* (*suffix*) of x.

The set of all distinct nonempty substrings of x (*words*) is called the *vocabulary* of x and denoted by V_x. Two factors $x[i:j]$ and $x[m:n]$ are *equivalent* if their associated substrings are identical.

Let \$ be a special symbol not included in the alphabet I. A data structure suitable for organizing the words in V_x is the so called *suffix tree* [14] T_x for x \$. As is well known, such a tree T_x is rooted, has $O(n)$ nodes and for a string x \$ is defined as follows. Each arc is associated with a word in V_x by means of a suitable factor of $x[0:n]$, and each path from the root to a leaf describes the suffix obtained by concatenating the substrings associated with the sequence of its arcs. Thus, if x \$ is stored in $x[0:n]$, a leaf of T_x is labelled with the integer j if the corresponding path describes the suffix $x[j:n]$. An arc is labelled by an ordered pair (i, j) $(i \leq j)$

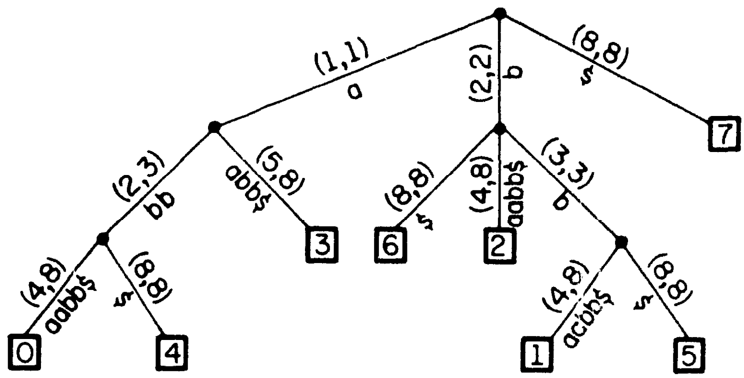

Fig. 1. The suffix tree of the string $abbaabb\$$.

if the associated substring is identical to the substring of the factor $x[i:j]$ (see for an example Fig. 1).

Although a brute force approach would use $O(n^2)$ operations to construct T_x for $|x| = n$, there exist clever algorithms for its construction in linear time [1, 14, 17].

Any vertex α of T_x distinct from the root describes a substring $W(\alpha)$ of x in a natural way (the concatenation of the factors associated with the arcs leading to α from the root): vertex α is called the *proper locus* of $W(\alpha)$. In general, for any $w \in V_x$, the *locus* α of w is the unique vertex of T_x such that w is a prefix of $W(\alpha)$ and $W(\text{FATHER}[\alpha])$ is a proper prefix of w. It follows from the definition of T_x, that for any substring w of x whose locus is α, the number of distinct occurrences of w in x (the number of equivalent factors associated with w) is equal to the number of leaves of the subtree of T_x rooted at α. In addition, the labels of the leaves of this subtree completely identify the positions of the first symbols of all factors whose substrings are identical to w.

Finally, we recall that a string $x \in I^+$ is *primitive* if setting $x = u^k$ implies $u = x$ and $k = 1$. It is a simple exercise to show that with the aid of the suffix tree we can decide in linear time if a string is primitive (or any of its prefixes is not). A string $x \in I^+$ is *strongly primitive* or *square-free* if, expressing x as $x = v_1 u^k v_2$, with $u \in I^+$ and $v_1, v_2 \in I^*$, implies $k = 1$. Equivalently, x is square-free if and only if each $w \in V_x$ is primitive.

To decide whether a string is square-free is a more complicated problem. It is easy to see, however [10], that x is not square-free if and only if there are equivalent factors $x[i:j]$ and $x[l:m]$, with $l > i$, such that $l \leq j+1$. Let vertex α of T_x be the common locus of the word w associated with these two equivalent factors; then the subtree of T_x rooted at α contains both leaves labeled i and l. Since $l - i \leq j + 1 - i = |x[i:j]| \leq |W(\alpha)|$, we can state the following straightforward theorem:

Theorem 1. *A string x is not square-free if and only if there is at least one interior vertex α of T_x such that $|W(\alpha)|$ is not greater than the difference of the labels of any two leaves of the subtree rooted at α.*

Assume that x is not square-free. A *repetition* in x is a factor $x[i:m]$ for which there are indices l and j ($i<l\leqslant j\leqslant m$) such that:

(i) $x[i:j]$ is equivalent to $x[l:m]$,

(ii) $x[i:l-1]$ corresponds to a primitive word and

(iii) $x[j+1]\neq x[m+1]$.[1]

We recall that p is a period of w if $w[i]=w[i+p]$ ($\forall\, i=1,2,\ldots,|w|-p$). It is easily seen [10] that a repetition is a positioned periodic substring in the form $(st)^k s$, where $k>1$, $s\in I^*$, $t\in I^+$, which is completely identified by the triple $(i, l-i, m-i)$ of its starting position, its period, and its length, respectively. It follows from points (i) and (iii) in the above definition that there must be a vextex α in T_x such that $W(\alpha)$ corresponds to $x[i:j]$. We now claim that i and l must be consecutive integers in the set of integers associated with the leaves of the subtree of T_x rooted at α.

In demonstrating our claim, we make use of the following well-known 'periodicity' lemma [10]:

Lemma 1. *If w has periods p and q and $|w|\geqslant p+q$ then w has period g.c.d. (p,q).*

Now, let $x[i:m]$ be a repetition; it has the form $(st)^k s$, where st is its primitive periodic part, with $|st|=(l-i)\triangleq p$. Assume now that in the subtree of T_x rooted at α there is some other leaf, labeled b, with $i<b<l$. Since $(b-i)<(l-i)\leqslant|W(\alpha)|$, there is another repetition, starting at position i, of the form $(uv)^r u$, with periodic part uv (where $|uv|=(b-i)\triangleq q$). Since $|W(\alpha)|\geqslant|st|$ and $|(uv)^r u|\geqslant|W(\alpha)|+|uv|\geqslant|st|+|uv|=p+q$, the factor $x[i:i+p+q-1]$ is a prefix of $(uv)^r u$. This factor has periods p and q and length $p+q$, whence, by the above lemma, it has also period g.c.d. $(p,q)\leqslant q<p$, which means that st is not primitive, a contradiction.

The above characteristic condition provides an algorithmic criterion. The process could be easily organized as a bottom-up computation. Starting from the leaves of T_x, for each interior vertex α visited we construct the sorted list of the labels of its leaves, compute the differences of consecutive labels and compare them with $|W(\alpha)|$. The sorted list for any such vertex is obtained by merging the sorted lists of its offspring vertices. Using 'natural merging' [9, p. 162] this strategy is certainly efficient if the suffix tree is nearly balanced and runs in such case in time $O(n\log n)$. Similarly, if the suffix tree is highly unbalanced and has a comb-like structure, the above strategy results in the same performance (since each 'merge' becomes the insertion of a single element into a heap).[2] Despite the simplicity of these two extreme cases the intermediate cases are more difficult to handle. In the next section we shall present a data structure, the leaf-tree, which supports the outlined strategy

[1] For convenience, we give here a definition of repetition that slightly differs from the one usually found in the literature: there, the definition refers to strings in the form u^k rather than their longest extensions $u^k u'$ (with u' a prefix of u).

[2] Actually, it takes constant time to detect a comb-like structure on line with the construction of T_x: since the path for $x[i:n]$ in T_x is a tributary of that for $x[i+1:n]$, then it must be $x=v_1 a^k v_2$ for some $a\in I$, $k>1$ and $v_1, v_2\in I^*$.

in time $O(n \log n)$ irrespective of the structure of the suffix tree. For the sake of clarity, we shall present the leaf-tree in two steps: at first in a version which has a total memory usage $O(n \log n)$; subsequently, we transform the implementation of the leaf-tree to a more compact data structure, using just $\theta(n)$ (i.e., optimal) space in the overall execution of the algorithm.

3. Leaf-trees

We introduce now a data structure which is suited for merging two disjoint sorted sequences S_1 and S_2. Each such sequence here and hereafter is a subsequence of the sequence $(0, 1, \ldots, n-1)$.

The *leaf-tree* $T(S)$ associated with a given integer sequence S is a composite data structure which supports sequential and binary search access to the elements of S. T contains a strictly tree-like portion and a linear list portion as its main components. Vertices of the tree portion will be called *nodes*, to avoid confusion with the suffix-tree discussed in Section 2.

At this point in the presentation, we shall think of the leaf-tree $T(S)$ as a standard balanced binary tree with n leaves (and a total of $2n-1$ nodes). The structure is static, that is, independently of S each node is identified with a unique storage area and thus the pointers from a node to its offspring nodes are implicit in the storage allocation. The leaves of $T(S)$ may be viewed as an array (this suggestion is made only to order the leaves, not to make use of the random access properties of an array), and *active* leaves are the positions of this array corresponding to the elements of S. The linear list portion of $T(S)$ threads the active leaves in ascending order.

Given a node V in the tree-portion of $T(S)$, by TREE(V) we denote the subtree rooted at V and by LSON(V), RSON(V) we denote the left and right offsprings of V respectively. Let m_V and M_V represent the smallest and largest values, respectively, of leaves which are currently active in TREE(V). We associate with each node V two fields, min[V] and max[V], whose contents we now define. Any time in $T(S)$ there is a leaf-bound path $V_1, V_2, \ldots, V_{k-1}, V_k$ with $(m_{V_1}, M_{V_1}) \neq (m_{V_2}, M_{V_2}) = \cdots = (m_{V_{k-1}}, M_{V_{k-1}}) \neq (m_{V_k}, M_{V_k})$ and $k \geqslant 5$, we set up a BYPASS pointer from V_2 to V_{k-1} which effectively compresses the path $V_2 \cdots V_{k-1}$ to its two terminal nodes; nodes $V_3, \ldots, V_{k-2}$ are given the status of 'bypassed'. Thus we have

$$(\min[V], \max[V]) = \begin{cases} (\Lambda, \Lambda) & \text{if either TREE}(V) \text{ is empty or } V \text{ is 'bypassed'}, \\ (m_V, M_V) & \text{otherwise.} \end{cases}$$

A node V for which $(\min[V], \max[V]) \neq (\Lambda, \Lambda)$ is referred to as *active* (*inactive* otherwise). If we now consider the tree formed solely by active nodes, we note that this tree has a number of leaves equal to the cardinality of S and a number of internal nodes which is always less than three times the number of leaves.

The linear list portion of $T(S)$ is straightforward and is described by the pointer NEXT[ ]. However, $T(S)$ is completed by an array of pointers $L[0:n-1]$ associated with the leaf array defined as follows:

$$L[i]=\begin{cases} \Lambda & \text{if leaf } i \text{ is not active,}\\ U & \text{the highest node in the path from the root of } T(S)\\ & \text{to leaf } i \text{ such that } \min[U]=i.\end{cases}$$

In addition, at each node U, such that $L[i]=U$, we have a backward pointer $L^{-1}[U]=i$; $L^{-1}[U]=\Lambda$ when no i points to U (U is inactive).

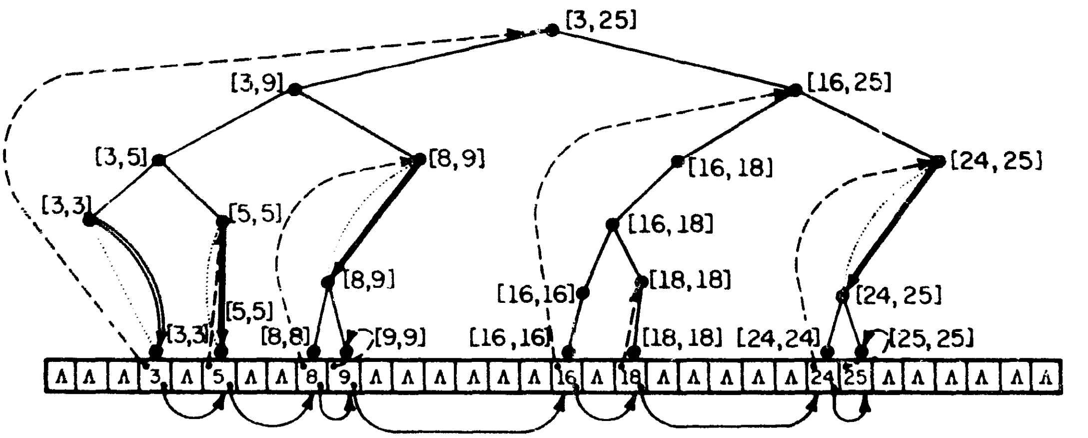

Fig. 2. An example of a leaf-tree. Active nodes are shown solid; L-pointers are shown as broken lines, bypass pointers as double lines, bypassed paths as dotted lines, and solid lines thread the list of active leaves (NEXT[ ]).

The reader who is getting impatient at the description of an apparently clumsy object, may find relief in perusing Fig. 2 where $T(S)$ is illustrated for $S=\{3,5,8,9,16,18,24,25\}$. Only active nodes are displayed, and different graphical lines are used for the various types of pointers. L-pointers are shown by broken lines, bypass pointers by double lines, dotted lines denote bypassed paths, and solid lines are used for the threaded list on the leaf array.

Note that, according to the definition, each active L-pointer (except the one pointing to the root) is directed to an RSON node of the tree. For uniformity we may assume that the root is itself the RSON of a dummy node.

As noted earlier, when the number of active leaves is substantially smaller than n, very few nodes of $T(S)$—both leaves and internal nodes—are actually used. This apparently wasteful realization of the leaf-tree has the following important property: given two such trees $T(S_1)$ and $T(S_2)$, due to the fixed storage allocation, it is possible to access in constant time the tree-node of $T(S_1)$ homologous[3] of a selected tree-node of $T(S_2)$, and *vice versa*. We shall see later, however, that this behavior can be emulated by a more subtle and compact implementation of the leaf-tree,

[3] I.e., equally placed in the data structure.

whose description is deferred in order to separate functional aspects from issues of efficiency.

All operations on leaf-trees can be interpreted as the merging of two sorted sequences. If $|S^{(1)}| \geq |S^{(2)}|$, we shall always merge $S^{(2)}$ into $S^{(1)}$. This merge is done by *inserting* the terms of $S^{(2)}$, one at a time and in sorted order, into $S^{(1)}$. So, 'insertion' is the primitive operation. From the data structure standpoint, merging is effected by operating on the leaf-trees $T(S^{(1)})$ and $T(S^{(2)})$, by performing an in-place update of $T(S^{(1)})$. We shall now analyze the mechanics of this update.

For notational identification, superscripts (1) and (2) denote entities in $T(S^{(1)})$ and $T(S^{(2)})$, respectively; also the absence of a superscript denotes an entity which has been updated—in $T(S^{(1)})$—to its final status, i.e., the status attained in $T(S^{(1)} \cup S^{(2)})$. With each term i of $S^{(2)}$ we associate the set of nodes, TREE$[i]$, of the subtree whose root is pointed to by $L^{(2)}[i]$ in $T(S^{(2)})$; note, that this is a uniquely specified set of nodes, independently of whether we are considering them in $T(S^{(1)})$ or $T(S^{(2)})$.

We shall now describe, in great detail, the procedure *insert*. Our claim is as follows:

Theorem 2. *The term-by-term insertion of $S^{(2)}$ into $S^{(1)}$ by means of procedure 'insert' correctly transforms $T(S^{(1)})$ into $T(S^{(1)} \cup S^{(2)})$.*

Proof. We shall need the following lemma:

Lemma 2. *Prior to the insertion of $i \in S^{(2)}$ into $S^{(1)}$, the following nodes of $T(S^{(1)})$ have been updated to their final status:*

(a) *on the path from the root to leaf i, all nodes preceding the root U of* TREE$[i]$; *this is referred to as* PATH(i) (*see Fig.* 3);

(b) *all the nodes in the left subtrees of the nodes specified in* (a).

In addition all the right subtrees of the nodes of PATH(i) *are legitimate leaf-trees* (*of substrings of $T(S^{(1)})$*).

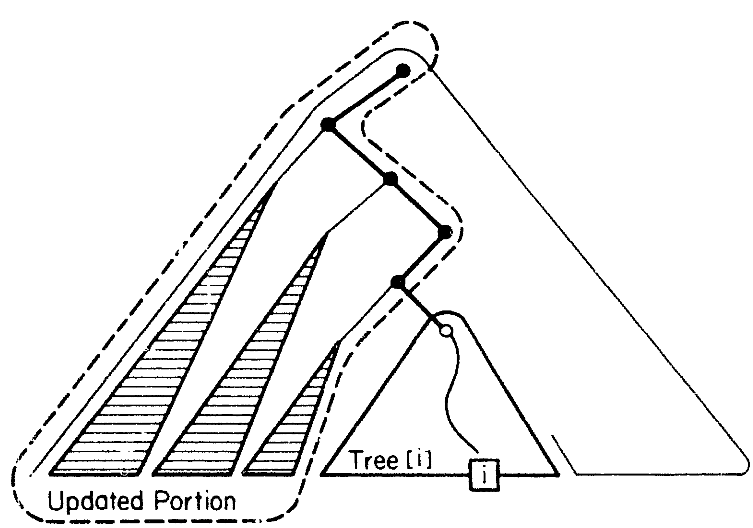

Fig. 3. Updated portion of T prior to the insertion of i. The nodes of PATH(i) are shown solid.

Indeed, Lemma 2 implies that after the insertion of the *largest* term j of $S^{(2)}$ all the left-subtrees of PATH(j) are in their final status, and so are the right-subtrees (if any), which do not contain any term of $S^{(2)}$. □

Proof of Lemma 2. We note at first that the conditions of the lemma are trivially satisfied when the first (smallest) term i_0 of $S^{(2)}$ is to be inserted: indeed, TREE[i_0] is the entire leaf-tree and the set of updated nodes is empty. Next, assuming it to be satisfied prior to the insertion of $i \in S^{(2)}$, we must show it still holding prior to the insertion of NEXT[i]$\in S^{(2)}$. Referring to Fig. 3, the extension of the inductive hypothesis is immediate if TREE[NEXT[i]] is the right subtree of a node in PATH(i), defined above; so we only need to restrict our attention to the case when TREE[NEXT[i]]$\subset$TREE[i]. Letting $U^{(2)} = L^{(2)}[i]$, the insertion of i is effected by the following procedure (i remains a global variable for the procedure):

```
proc insert(U, i)
begin l ← i
  if L⁻¹[U⁽¹⁾] ≠ Λ then advance(U, l)
  else begin L[i] ← U
             copy(U, l)
       end
end
```

Basically, two entirely different actions take place in TREE[i] depending upon whether node $U^{(1)}$, the root of TREE[i] in $T(S^{(1)})$, is also the destination of an L-pointer.

In the negative case, no leaf of TREE[i] is active in $T(S^{(1)})$, so that the content of TREE[i] in $T(S^{(2)})$ must be copied (active leaves and nodes) into the homologous positions of T. Since $T(S^{(2)})$ is by hypothesis a leaf-tree, the inductive hypothesis is trivially extended. The copy is actually carried out path by path, that is, all the active nodes from U toward leaf i are copied into T by the following straightforward procedure (here INTER[U]$=$(min[U], max[U]), BYPASS[U] is obviously the bypass pointer at U, and SON(U, i) is the son of node U in the direction of leaf i):

```
proc copy(U, i)
begin INTER[U] ← INTER[U⁽²⁾]
  BYPASS[U] ← BYPASS[U⁽²⁾]
  if (U is not a leaf) then
    begin if BYPASS[U] ≠ Λ then V ← BYPASS[U]
          else V ← SON(U, i)
               copy(V, i)
    end
end
```

An illustration of the working of copy is given in Fig. 4. Here active leaves are denoted by the symbol □, active nodes are solid and labeled with an upper-case

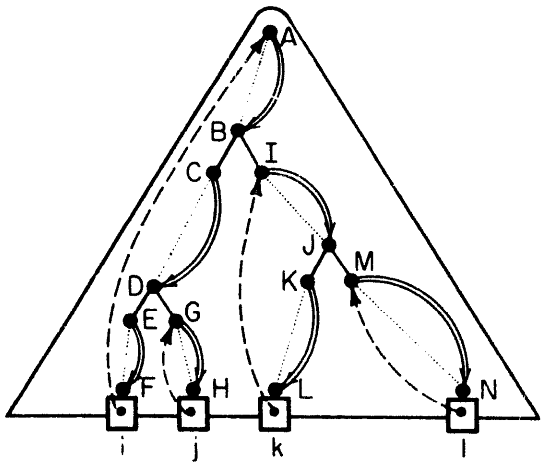

Fig. 4. Illustration of the action of procedure *copy*.

letter. $copy(A, i)$ copies nodes A, B, C, D, E, F; $copy(G, j)$ copies G and H; $copy(I, k)$ copies I, J, K, L; $copy(M, l)$ copies M and N. Note that the use of BYPASS links is essential to guarantee that, if *copy* has to transfer m active leaves from $T(S^{(2)})$ to T, at most $4m-2$ nodes will have to be copied.

We now consider the case in which there are active leaves of TREE[i] in $T(S^{(1)})$, which is somewhat more complicated. Informally, we initially have two leaves i and k pointing to the same node U; the final result will be that $\min(i, k)$ will point to U, while for $\max(i, k)$ we shall trace and process (part of) its leafward path from U until the appropriate final destination of $L[\max(i, k)]$ is found on this path; let $P(U, i)$ denote the nodes traversed in this leafward march. Note that:

(1) Since i is the smallest term of $S^{(2)}$ in TREE[i] (i.e., the smallest label of the active leaves of $S^{(2)}$ in TREE[i]) only terms of $S^{(1)}$ may affect the left subtrees of nodes of $P(U, i)$; thus no update is due in these left subtrees. As to the right subtrees, we shall illustrate below that at most one of them may require processing of its root to verify the lemma (referred to as 'special case', below).

(2) If NEXT[i] is a leaf of TREE[i], it is also a leaf of a right subtree of a node $V \in P(U, i)$; thus if all nodes of $P(U, i)$ have been updated to their final status, since all nodes from the root of T to U had been (inductively) updated, Condition (a) of Lemma 2 will be met for NEXT[i].

Therefore, updating the status of the nodes of $P(U, i)$, and taking care of the 'special case' mentioned above, would extend the inductive hypothesis and would prove the correctness of our insertion procedure.

To facilitate the understanding of the basic ideas of *advance* we now introduce a pedagogical simplification, to be waived with no penalty in the complete description of the procedure given in the Appendix. This simplification consists in disabling all BYPASS links, i.e., in assuming that all nodes of a leaf-tree are active. Since the central action of *advance* is the leaf-ward migration of L-pointers, the introduced simplification avoids unnecessary cluttering of the procedure. We shall later justify that the complete procedure has essentially the same performance as the simplified version to be now described. Recall that i—the leaf to be inserted—is a global variable for the following procedure *advance*(U, l), and that initially $l = i$.

```
        proc advance(U, l)
1.      begin if (U ≠ Λ) then (* otherwise the procedure is aborted *)
2.        begin INTER[U] ← INTER[U^(1)] ∪ INTER[U^(2)]
3.          if (U is RSON) then
4.            begin k ← L^(-1)[U^(1)]
5.              if (k = Λ) and (min[U] = l) then
6.                begin terminate(U, l)
                      U ← Λ
                end
7.              else if (k > l) then
8.                    begin L[l] ← U (* see footnote 4 *)
9.                          l ← k
                    end
            end
10.         if (U ≠ Λ) then U ← SON(U, l)
11.         advance(U, l)
          end
        end
```

Processing starts at the root of TREE$[i]$ and proceeds toward a leaf, since *advance*(U, l) issues a call *advance*(SON(U, l), l) in step 11. This march terminates in step 5, when the procedure *terminate*(U, l), to be discussed below, completes processing and sets $U \leftarrow \Lambda$; the subsequent call *advance*(Λ, l) aborts the march.

Step 2 performs the update of the interval of U. Since an LSON node cannot be the destination of an L-pointer, processing of an LSON reduces to the interval update. When processing an RSON, however, we must check whether that node is already the destination of an L-pointer (steps 4, 5, 7), and, if so, select the larger of the two labels pointing to the node and make its pointer migrate leafward (see steps 8 and 9, where k assumes the role previously held by i). Note that, while being inserted, term i may encounter as many as $\lceil \log_2 n \rceil$ L-pointers, but that the change of role (steps 8 and 9) may occur at most once. Indeed, this happens at the node V where the paths toward i and k diverge and $i < k$. In this case, k begins its leafward migration. On the other hand, since k is smaller than any remaining element in the subtree (recall that $L^{(1)}[k]$ pointed to the root of TREE$[i]$), it cannot dislodge any other L-pointer and the migration stops at the RSON of V. This is the only processing of right subtrees of the $P(U, i)$, the special case we alluded to before.

[4] We are implicitly assuming here that the address to leaf l in $T(S^{(1)})$ is available, so that $L[l]$ can be assigned. This is trivial if the leaves form an array. However, we shall see later that is just as simple when only the list NEXT is available.

Finally, we discuss the procedure *terminate* (step 5). When we reach an RSON U for which $L^{-1}[U^{(1)}]=\Lambda$ (i.e., U is not pointed to in $T(S^{(1)})$) and $\min[U]=l$, then clearly we have reached the destination of $L[l]$. We have two cases (recall that i is the term being inserted):

(a) $l=i$. In this case, after directing the L-pointer of i to U, since there are no elements of $S^{(1)}$ in the tree rooted at U, further processing reduces to a simple copy operation.

(b) $l=k$. In this case, the paths towards leaves i and k have diverged at FATHER[U], with SON(FATHER[U], i) being the LSON. Thus, after directing the L-pointer of k to U, processing is completed by a copying operation starting at the left sibling of U (indeed, in the subtree rooted at this node there is no term of $S^{(1)}$).

With this premise, we have:

> **proc** *terminate*(U, l)
> **begin** $L[l] \leftarrow U$ (* see footnote 4 *)
>   **if** $(l \neq i)$ **then** $U \leftarrow$ LSIBLING $[U]$
>   **copy**(U, i)
> **end**

This completes the proof of Lemma 2. □

We now analyze the performance of the above procedure when merging two sorted sequences $S^{(1)}$ and $S^{(2)}$. In general, we may charge the computational work to each call of *copy* and *advance*. Each such call is executed in time bounded by a constant, as may be easily seen by inspection of the two detailed procedures. When merging $S^{(2)}$ into $S^{(1)}$, *copy* may be called at most as many times as there are active internal nodes in $T(S^{(2)})$; but we know that the number of the latter is less than three times $|S^{(2)}|$, whence the work attributable to *copy* is proportional to $|S^{(2)}|$. Again, note the crucial importance of the BYPASS pointers to assure the latter result. If we now consider that we always merge a shorter sequence into a longer one, each term in $\{0, \ldots, n-1\}$ is involved in a merge-into process at most $\lceil \log_2 n \rceil$ times, whence the total amount of work attributable to *copy* when successively merging disjoint subsets of $\{0, \ldots, n-1\}$ is bounded by $O(n \log n)$. Similarly, the work attributable to *advance* is measured by the number of nodes visited by L-pointers in their leafward migration. Since each L-pointer starts at the root and can only descend toward a leaf, the number of visited nodes is bounded by $\lceil \log_2 n \rceil$, whence also the work attributable to *advance* is bounded by $O(n \log n)$.

4. Application of leaf-trees to the detection of repetitions

The leaf-tree, and its associated handling procedure, as described in Section 3 can now be used for the detection of repetitions in a given textstring x.

A preliminary step, of course, consists of constructing the suffix-tree T_x of x; we have already recalled that this task runs in time $O(n)$ [14], where $n = |x|$. The suffix-tree, in general, has maximum node degree equal to $|I \cup \{\$\}|$; we transform it into a binary tree in a straightforward way by adding appropriate dummy arcs and vertices. A dummy arc is associated with the empty symbol Λ and a dummy vertex is identical to his father (i.e., for a dummy β, $W(\beta) = W(\text{FATHER}[\beta])$). The resulting structure is referred to as the *modified suffix-tree.*

We now visit the vertices of the modified suffix-tree as in a pebbling game [15]. Specifically, we have a given number of 'pebbles' and visiting a vertex means to place a pebble on that vertex, where pebbling is subject to the following rules:

(i) a leaf can be pebbled unconditionally;

(ii) a nonleaf vertex can be pebbled if and only if its offsprings are both presently pebbled. (We adopt the convention to move the pebble from the left offspring to the father, while the other pebble is free and reusable.)

It has been shown that a tree with n leaves can be pebbled (i.e., all of its vertices can be pebbled) with $O(\log n)$ pebbles [15].

In our application the role of pebbles is taken by leaf-trees and pebbling a vertex of the modified suffix-tree corresponds to merging the two sequences associated with its offspring. In this operation, one of the two leaf-trees is updated while the other becomes reusable. We defer the analysis of the time and space requirements of this scheme until the illustration of a space-efficient implementation of the leaf-tree, to be given in the next section.

As we mentioned in Section 2, the objective of merging $S^{(1)}$ and $S^{(2)}$ was the calculation of the 'gaps' between elements of $S^{(1)}$ and of $S^{(2)}$, respectively. When inserting $i \in S^{(2)}$ into $S^{(1)}$, i will fall between two consecutive terms j and k; the values of $|i-j|$ and $|k-i|$ are obtained when adjusting the list links NEXT, as shown in the Appendix. The minimum gaps can thus be obtained and compared with $|W(\alpha)|$. Note that, as we move rootward on a path of the tree T_x, the value of $|W(\alpha)|$, with which gaps are to be compared, decreases. Thus, a pair of terms $i \in S^{(2)}$ and $j \in S^{(1)}$ which did not generate an overlap when i was first inserted, need not be re-examined any further.

5. A space-efficient implementation of leaf-trees

Once the structure and functional capabilities of leaf-trees are well understood (Section 3), as well as their application to the detection of repetitions of substrings of a string (Section 4), we can tackle the problem of their space-efficient implementation.

As we mentioned in Section 3, the reason for choosing a statically allocated realization with $O(n)$ storage, rather than a linked structure realization with $O(|S|)$ storage, was the wish to execute with great ease the following operation:

"*Given $U^{(2)}$ in $T(S^{(2)})$ find the corresponding $U^{(1)}$ in $T(S^{(1)})$.*"

Clearly this operation is trivial in the proposed realization since the addresses of $T(S^{(1)})$ are just a translation of those of $T(S^{(2)})$.

We now want to show that this behavior can be emulated in a linked-structure realization of leaf-trees, with the aid of an additional data structure $\mathscr{D}$, called *directory*. The method is admittedly quite complicated, but, nevertheless, it exhibits asymptotically efficient storage utilization.

While a leaf-tree is realized as a linked-structure, the directory $\mathscr{D}$ has the same storage allocation as the leaf-tree defined in Section 3, i.e., it is a balanced binary tree with $2n-1$ nodes, each with random-access capabilities. Each node U of a leaf-tree $T(S)$ corresponds to its homologous U^* in $\mathscr{D}$, that is, from U we can access U^* in constant time (either through a pointer, or by random access on the basis of U's name). Suppose now that, during the execution of the algorithms there are k active leaf-trees, $T(S^{(1)}),\ldots,T(S^{(k)})$. Associated with each U^* of $\mathscr{D}$ there is a collection of pointers to $\{U^{(j)}: j\in\{1,2,\ldots,k\}$ and $U^{(j)}$ is active$\}$, i.e., to its active homologous nodes in the leaf-trees. We now show that each of these collections can be organized as a stack as a consequence of the following strategy:

(i) The modified suffix-tree T_x is rearranged so that for each internal vertex the left subtree contains no fewer leaves than the right subtree;

(ii) Pebbling of the modified suffix tree, rearranged as specified above, is done according to a *post-order visit* of the vertices [9].

Condition (i) insures that it will always be the leaf-tree corresponding to a right-son of T_x that is merged into the one corresponding to its left sibling (recall that the lighter of two leaf-trees is merged into the heavier one). Next, imagine to have a hypothetical structure, called STACK (with conventional PUSH and POP operations, denoted, respectively by 'STACK $\Leftarrow$' and '$\Leftarrow$ STACK') to be used in conjunction with the visit of T_x. We now give a concise description of the overall algorithm, where the operation 'STACK $\Leftarrow \alpha$' is to be interpreted as the construction of the leaf-tree associated with the vertex α of T_x:

begin STACK $\leftarrow \phi$
  while there are vertices of T_x to be visited **do**
    begin $\alpha \Leftarrow$ get vertex in post-order visit of T_x
      if (α is a leaf) **then** STACK $\Leftarrow \alpha$
      if (α is a right-son) **then**
        begin $\alpha \Leftarrow$ STACK
              $\beta \Leftarrow$ STACK
              STACK $\Leftarrow$ FATHER(α,β)
        end
    end
end

Clearly, STACK contains a sequence of leftsons in T_x, possibly terminated at the top with a single rightson. As a consequence of the visiting policy (ii) and of the above algorithm, any time we reach a rightson vertex α, STACK contains α and its left sibling β in its two top positions: they are popped and replaced with their father.

The hypothetical STACK is mirrored by a corresponding data structure STACK(U^*) for each node U^* of the directory $\mathscr{D}$. Specifically, let $(\alpha_1, \alpha_2, \ldots, \alpha_p)$ be the sequence of terms in STACK and let $\mathscr{T}_k$ be the leaf-tree pertaining to vertex α_k of T_x. For a node U^*, STACK(U^*) contains the set of pointers to $\{U^{(k)}$: $U^{(k)}$ homologous to U^* in $\mathscr{T}_k$ and active$\}$ as a *sequence* ordered according to the index k. Therefore, assume that TOP(STACK)$=\alpha_p$; if $U^{(p)}$ is active in $\mathscr{T}_p$, then TOP(STACK(U^*)) is a pointer to $U^{(p)}$. On the other hand, TOP(STACK(U^*)) does not point to $U^{(p)}$, if the latter is inactive. Thus, to test whether TOP(STACK(U^*)) really points to $U^{(p)}$, it is sufficient to have each active node in leaf-trees point to a *tree designator* containing the name of the leaf-tree. This device not only enables the test just described, but in addition it proves crucial to the efficiency of the algorithm. Indeed, when merging $T(S^{(2)})$ into $T(S^{(1)})$ (see Section 3), $T(S^{(1)})$ is updated to $T(S^{(1)} \cup S^{(2)})$; all the nodes of $T(S^{(1)})$ which are not visited by the merging task have their tree membership collectively updated by the single update of the tree designator.

In summary our original task, i.e., the operation of obtaining $U^{(1)}$ from $U^{(2)}$, is pictorially described in Fig. 5. The sequence of links is self-explanatory. Note that, due to condition (i) above, all pointers to nodes of $T(S^{(2)})$ are popped from the corresponding stacks in the directory before actual merging begins.

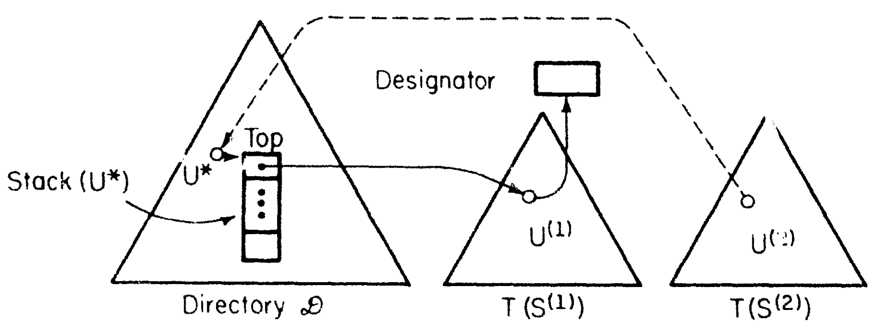

Fig. 5. Sequence of links to access $U^{(1)}$ (if active) from $U^{(2)}$. If the designator is not that of $S^{(1)}$, then $U^{(1)}$ is inactive.

We can now return to the analysis of the time and space requirements of the proposed scheme. Leaf-tree $T(S)$ uses storage $O(|S|)$, whence the total storage used by all leaf-trees active at any one time (those corresponding to the vertices of T_x contained in STACK) is $O(n)$. Storage space is assigned to the linked structures and reusable space ('garbage') is collected in standard fashion. The work pertaining to recycling this memory space can be charged to each *insert*, i.e., the latter work is increased by a bounded amount; since the total work of *insert* is $O(n \log n)$, so is the space recycling work. Thus, we conclude with the following theorem:

Theorem 3. *The detection of repetitions in a textstring of length n can be carried out in time $O(n \log n)$ and space $O(n)$.*

The time bound has been shown to be optimal by Crochemore [4]; the space bound is trivially optimal.

Appendix

Some significant additions to the listings of the procedures presented in Section 3 are necessary to account for two actions which were intentionally ignored in the preceding presentation:

(1) the management of the list of active leaves, via the pointer array NEXT. The steps corresponding to this task will be displayed within broken-line boxes for quick reference;

(2) the use of BYPASS links. The corresponding steps will be displayed within solid-line boxes.

The updating of NEXT occurs immediately after the assignment of a value to $L[i]$ (recall that i is the element of $S^{(2)}$ *actually* being inserted into $S^{(1)}$). This assignment occurs in one of three places:

(i) when $L^{-1}[U^{(1)}]=\Lambda$ (there is no term of $S^{(1)}$ in TREE[i]), within the procedure *insert* itself. Box I4–5 below describes the action: note that I4 correctly assumes that SIBLING[$U^{(1)}$] is active, for INTER[FATHER[$U^{(1)}$]] $\neq$ INTER[SIBLING[$U^{(1)}$]]. Since the largest term in T smaller than i is stored in the subtree rooted at the left-sibling of $U^{(1)}$, then this term is max[SIBLING[$U^{(1)}$]], and the corresponding update takes place;

(ii) when TREE[i] contains terms of $S^{(1)}$ and within *advance* there is a dislodgement of an L-pointer (see steps A8–11 below);

(iii) after *terminate* has been invoked (provided that $L[i]$, and not $L[k]$, is being assigned). Steps T3–4 show this action: note that since the tree rooted at U does not contain any term of $S^{(1)}$ (see step A6) then $M_{\text{FATHER}[U^{(1)}]}$ is the largest term smaller than i in T; since its interval differs from those of its offspring, FATHER[$U^{(1)}$] is active, whence $M_{\text{FATHER}[U^{(1)}]}=\max[\text{FATHER}[U^{(1)}]]$.

We now consider the handling of BYPASS links. It is convenient to provide a concise review of the actions of the various procedures *as described* in Section 3. Procedure *insert*(U, i) basically traces a path from node U (the root of TREE[i]) to leaf i (see Fig. 6). In Fig. 6 the nodes shown as solid circles are those visited by *advance*, while those shown as empty circles are those visited by *copy*; the portion visited by *copy* is never empty, while that visited by *advance* may be empty. When both sets are nonempty (i.e., there is at least one element k of $S^{(1)}$ in TREE[i]), the node V where *advance* stops is also visited by procedure *terminate*. FATHER[V] is where the paths to leaf i diverges from the path to leaf k; if V is an LSON, then *terminate* visits also SIBLING[V].

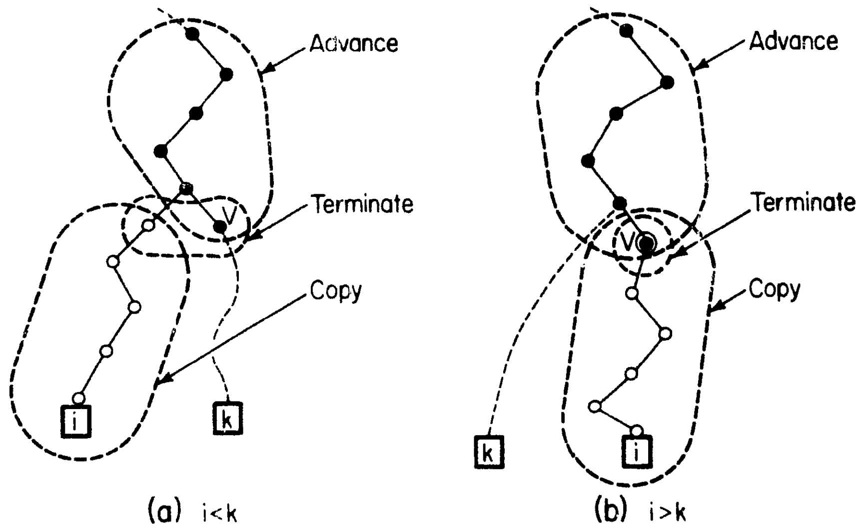

Fig. 6. Illustration of the nodes visited by *advance*, *terminate*, and *copy* in the two cases when $i<k$ or $i>k$.

Assume now that BYPASS links are used, and let path($U'\to V'$) denote the path from node U' and to node V'. Note first that a BYPASS link issuing from node U' on path($U\to$FATHER[(V)]) in $T(S^{(1)})$ must be directed to some node V' on path($U'\to$leaf k). Indeed, $(m_{U'}, M_{U'})=(m_{V'}, M_{V'})$ by the definition of BYPASS, and $m_{U'}=k$, since k is the smallest term of $S^{(1)}$ in the subtree rooted at U; it follows that $m_{V'}=k$, i.e., V' is on path ($U'\to$leaf k). Analogously, in $T(S^{(2)})$ the destination V' of the BYPASS link is on path ($U'\to$leaf i).

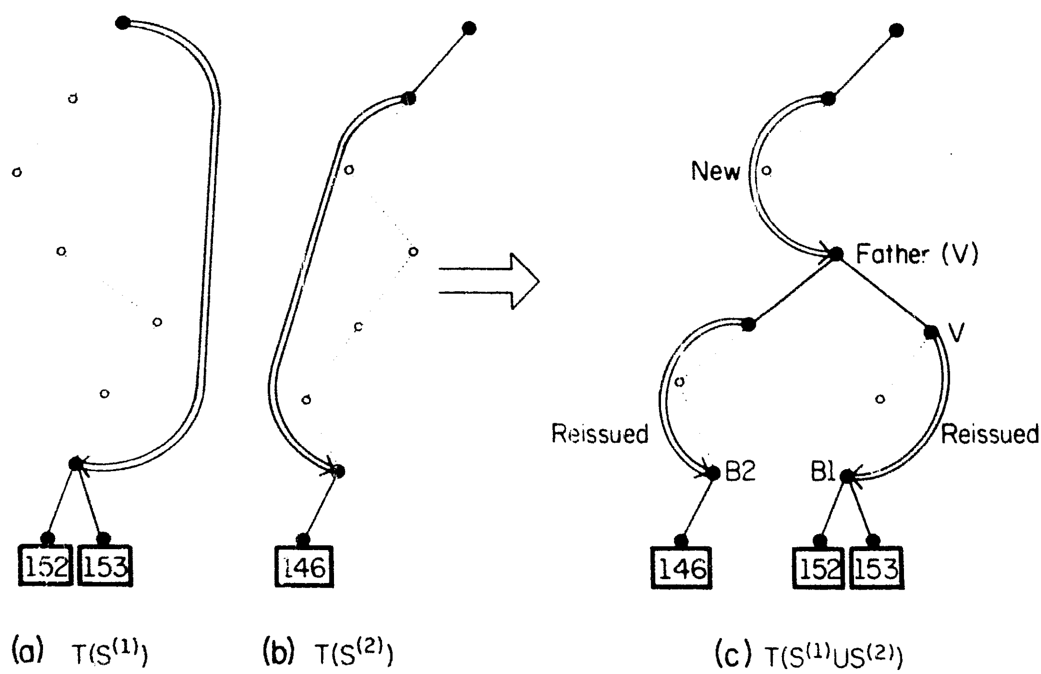

Fig. 7. Insertion of $i=146$ into $S^{(1)}$.

One approach to the handling of BYPASS links is the following (although a more compact approach is possible):

(i) Let *advance* trace path($U \to V$), node by node as in Section 3, ignoring BYPASS links except those (at most two) whose destination is beyond V and SIBLING[V]: indeed, any such BYPASS link may have to be reissued either from V or from its sibling.

(ii) trace path($U \to$FATHER[V]) *backward*, changing nodes to the inactive status and establishing BYPASS links as appropriate, in a straightforward manner.

We shall now elaborate on part (i). Let U^* be the node currently visited by *advance*. If $U^{*(i)}$ is inactive $(i = 1, 2)$, then clearly INTER[$U^{*(i)}$] = INTER[FATHER[$U^{*(i)}$]]: this provides an immediate means of reconstructing INTER[$U^{*(i)}$], in case $U^{*(i)}$ had become inactive during previous processing. At the same time, we save the destination of BYPASS links issuing from the visited nodes both in $T(S^{(1)})$ and $T(S^{(2)})$. Once we reach V, we test whether any of these destinations is beyond V and/or its siblings, and if so, reissue the appropriate BYPASSES. Figure 7 displays an example of insertion with reissue and creation of BYPASS links. The actions described are clearly shown boxed in the complete listings of procedures *advance* and *terminate*:

```
        proc insert(U, i)
I1      begin l ← i
I2        if L⁻¹[U⁽¹⁾] ≠ Λ then advance(U, l)
I3        else begin L[i] ← U
I4              NEXT[i] ← NEXT[max[SIBLING[U⁽¹⁾]]]
I5              NEXT[max[SIBLING[U⁽¹⁾]]] ← i
I6              copy(U, i)
             end
        end

        proc advance(u, l)
A1      begin  B1 ← B2 ← Λ
A2        if (U ≠ Λ) then (* otherwise the procedure is aborted *)
A3         begin INTER[U] ← INTER[U⁽¹⁾] ∪ INTER[U⁽²⁾]
A4          if (U is RSON) then
A5           begin k ← L⁻¹[U⁽¹⁾]
A6             if (k = Λ) and (min[U] = l) then
A7              begin terminate(U, l)
                       U ← Λ
              end
A8             else if (k > l) then
```

```
A9          begin L[l] ← U
A10               NEXT[l] ← NEXT[k]
A11               NEXT[k] ← l
A12               l ← k
            end
          end
A13      if (U ≠ Λ) then U ← SON(U, l)
A14      if (U^(1) is active) then
           if (BYPASS[U^(1)] ≠ Λ) then B1 ← BYPASS[U^(1)]
A15      else INTER[U^(1)] ← INTER[FATHER[U^(1)]]
A16      if (U^(2) is active) then
           if (BYPASS[U^(2)] ≠ Λ) then B2 ← BYPASS[U^(2)]
A17      else INTER[U^(2)] ← INTER[FATHER[U^(2)]]
A18      advance(U, l)
        end
      end

      proc terminate(U, l)
T1    begin L[l] ← U
T2     if (l = i) then
T3       begin NEXT[l] ← NEXT[max[FATHER[U^(1)]]]
T4             NEXT[max[FATHER[U^(1)]]] ← l
T5             INTER[SIBLING[U]] ← INTER[FATHER[U^(1)]]
         end
T6     else U ← SIBLING[U]
T7     if (B2 is below U) then BYPASS[U] ← B2
T8     if (B1 is below SIBLING[U]) then BYPASS[SIBLING[U]] ← B1
T9     copy(U, l)
      end
```

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley, Reading, MA, 1974).

[2] A. Apostolico, Fast applications of suffix trees, in: D.G. Lainiotis and N.S. Tzannes, Eds., *Advances in Control* (D. Reidel, Dordrecht, Netherlands, 1980) 558–567.

[3] C.H. Braunholtz, Solution to Problem 5030, *Ann. of Math.* **70** (1963) 675–676.

[4] M. Crochemore, An optimal algorithm for computing the repetitions in a word, *Information Processing Lett.* **12**(5) (1981) 244–250.

[5] J.P. Duval, Sur la périodicité des mots, Thèse de Docteur du 3me cycle, Faculty of Sciences, University of Rouen (1978).

[6] N.J. Fine and H.S. Wilf, Uniqueness theorems for periodic functions, *Proc. Amer. Math. Soc.* **16** (1965) 109–114.
[7] M.A. Harrison, *Introduction to Formal Language Theory* (Addison-Wesley, Reading, MA, 1978) 36–40.
[8] G.A. Hedlund, Remarks on the work of Axel Thue on sequences, *Nord. Mat. Tidskr.* **15** (1967) 148–150.
[9] D.E. Knuth, *The Art of Computer Programming, Vol 3: Sorting and Searching* (Addison-Wesley, Reading, MA, 1973).
[10] D.E. Knuth, J.H. Morris and V.R. Pratt, Fast pattern matching in strings, *SIAM J. Comput.* **6** (1977) 323–350.
[11] A. Lentin and M.P. Schützenberger, A combinatorial problem in the theory of free monoids, *Proc. University of North Carolina* (1967) 128–144.
[12] R.C. Lyndon and M.P. Schützenberger, The equation $a^M = b^N c^P$ in a free group, *Michigan Math. J.* **9** (1962) 289–298.
[13] M. Main and R. Lorentz, An O(n log n) algorithm for finding repetition in a string, T.R. 79-056, Computer Science Department, Washington State University, Pullman (1979).
[14] E.M. McCreight, A space economical suffix tree construction algorithm, *J. ACM* **23** (1976) 262–272.
[15] M. Paterson and C.E. Hewitt, Comparative schematology, *Proc. MAC Conference on Concurrent Systems and Parallel Computation*, Woods Hole, MA (1970) 119–127.
[16] A. Thue, Über die gegenseitige Lage gleicher Teile Gewisser Zeichenreichen, *Skr. Vid.-Kristiana I. Mat. Naturv. Klasse* **1** (1912) 1–67.
[17] P. Weiner, Linear pattern matching algorithms, *Proc. 14th Annual Symposium on Switching and Automata Theory* (1973) 1–11.