Theoretical Computer Science 92 (1992) 87–105
Elsevier

Optimal algorithms for computing the canonical form of a circular string*

Costas S. Iliopoulos

Royal Holloway College, University of London, Department of Computer Science, Egham TW20 0EX, England

W.F. Smyth

Department of Computer Science and Systems, McMaster University, Hamilton, Ontario, Canada L8S 4K1, and School of Computing Science, Curtin University, GPO Box U-1981, Perth WA 6001, Australia

Abstract

Iliopoulos, C.S. and W.F. Smyth, Optimal algorithm for computing the canonical form of a circular string, Theoretical Computer Science 92 (1992) 87–105.

An O(log n) time CRCW PRAM algorithm for computing the least lexicographic rotation of a circular string (of length n) over a fixed alphabet is presented here. The logarithmic running time is achieved by using O(n/log n) processors and its space complexity is linear. A second algorithm for unbounded alphabets requires O(log n log log n) units of time, also using O(n/log n) processors.

1. Introduction

Here we consider the question of determining the *least lexicographic rotation* (llr) of a *circular* string that yields the *canonical form* of the string. Formally, the problem can be stated as follows: given a circular string $x = a_1 \ldots a_n$ of length n over an alphabet Σ, compute an index k in the range $1 \leqslant k \leqslant n$ which satisfies the condition

$$a_k \ldots a_n a_1 \ldots a_{k-1} \leqslant a_i \ldots a_n a_1 \ldots a_{i-1} \quad \textit{for all } 1 \leqslant i \leqslant n.$$

*This work was partially supported by grant GR/E 75752 of the Science and Engineering Research Council of the UK. The authors were supported in part also by the UK SERC grant GR/F 00898 and a Royal Society grant, and by grant A8180 of the Natural Sciences and Engineering Research Council of Canada, respectively.

The string on the left-hand side is the llr of x, i.e., the original string rotated to the left by $k-1$ positions. One can see that k is not unique in the case where $x=y^l$ for some integer $l>1$, so that we have l possible starting points of the (always unique) llr.

The computational model used here is CRCW PRAM (Concurrent Read Concurrent Write Parallel RAM). The processors are unit-cost RAMs that can access a common memory. Some processors can access the same memory location: in fact, they can concurrently read and write; when two or more processors are attempting to write in the same memory location one of them succeeds in a nondeterministic fashion.

Let $Seq(n)$ be the best worst-case time complexity bound achieved by a sequential algorithm for the problem at hand, where n is the length of the input. Obviously, the best upper bound on the parallel time achievable using p processors without improving the sequential result is $(Seq(n)/p)$. A parallel algorithm that achieves this running time is said to have *optimal speed-up*, or simply said to be *optimal*.

Booth [6] gave a linear algorithm for computing the canonical form of a circular string by generalizing the Knuth–Morris–Pratt [10] linear time pattern-matching algorithm. Moreover, Shiloach [11] and Duval [7] gave another two linear algorithms improving the constant factor of the running time of Booth's algorithm (together with improvements on space complexity). All the three algorithms seem to be inherently sequential. In Booth's algorithm, the computation of the "failure function" is the main obstacle to parallelization, and in Shiloach's algorithm the series of comparisons is strictly sequential. Apostolico et al. [5] presented a prallel algorithm for the llr problem that requires $O(\log n)$ units of time and $O(n)$ processors to compute the canonical form of a string of length n over an alphabet of size $O(n)$. Given the linearity of the sequential complexity, there is obviously room for improvement of the parallel complexity bounds.

The algorithm for both the bounded and unbounded alphabets makes use of (i) *preprocessing* and (ii) a procedure for *duelling* between "locally tested" possible starting points of an llr.

(i) *Preprocessing*. This is necessary for reducing the problem to one of size $n/\log n$. In the constrained version of the problem, where the alphabet is fixed, we preprocess the input string by dividing it into blocks of size $\log n$ and computing a local starting point (by excluding points in the same block for which we have evidence that they are not starting points of an llr). Then we compute the set of all prefixes and suffixes of the substrings defined by two consecutive local starting points, each packed in a single word. In the case of unbounded alphabets we make use of suffix/prefix trees in order to speed up comparisons among the substrings involved.

(ii) *Duel*. The duel between two local starting points is dominated by the rule provided by the following lemma.

Lemma 1.1 (Apostolico et al. [5], Iliopoulos and Smyth [9]). *Let $w=a_i \ldots a_j$ and $z=a_j \ldots a_{2j-i}$ be substrings of a circular string x. If $w \leqslant z$ lexicographically, then a_j is a starting point of an llr if and only if both a_i and a_j are starting points of an llr.*

In the case of bounded alphabets the comparison of the two substrings required by Lemma 1.1 can be done in constant time using the packed words computed in preprocessing. In the case of unbounded alphabets we need a factor of $O(\log\log n)$ time to do this; we also make use of a PRIORITY CRCW PRAM (stronger version of the CRCW PRAM model which has all of its processors labeled with an integer so that when two or more processors attempt to write in the same memory location the one with the smallest label succeeds) in conjunction with the constant-time simulation procedure of the two versions of the model given in [8].

The results of this paper can be summarized as follows:

Theorem 1.2.[1] *Given a circular string x over a fixed alphabet, one can compute the canonical form of x in* $O(\log n)$ *units of time on a CRCW PRAM, with* $O(n/\log n)$ *processors requiring linear space.*

Theorem 1.3. *Given a circular string x of length n over an alphabet of size* $O(n)$, *one can compute the canonical form of x in* $O(\log n \log\log n)$ *units of time on a CRCW PRAM with* $O(n/\log n)$ *processors.*

2. Preprocessing over bounded alphabets

Given a string x we split it into blocks x_i, $1 \leqslant i \leqslant m-1$ of size $k = \lceil \log n \rceil$:

$$x = x_1 \dots x_{m-1} x_m, \quad m = \lceil n/k \rceil$$

with $|x_m| \leqslant k$. Furthermore, we compute the *local starting points* (lsp); i.e., a_s is said to be the lsp of $x_i = a_1, \dots, a_t$ if and only if s is the smallest index such that

$$a_s a_{s+1} \dots a_l \leqslant a_j \dots a_{j+l-s} \quad \text{for all } 1 \leqslant j \leqslant t,$$

where

$$x_{i+1} = a_{t+1} \dots a_{2t} \quad \text{and} \quad l = \begin{cases} j & \text{if } s \leqslant j, \\ 2s-j & \text{otherwise.} \end{cases}$$

(The lsp of x_m is defined as above using $x_{m+1} := x_1$ wrapping around the cyclic string). It is not difficult to see that the lsp of an x_i is just the starting point of the leftmost llr of all the rotations of the circular string $x_i x_{i+1}$ that start from a position within the block x_i. Now let $s_1, \dots, s_m$ be the lsp's of the blocks $x_1, \dots, x_m$, respectively. These are $O(n/\log n)$ possible starting points of an llr of x; the rest of the symbols of each block can be eliminated as candidates starting points of an llr by means of Lemma 1.1.

The s_i's split the string x into m substrings, whose prefixes and suffixes we encode ("pack") into single words. This is done in order to take advantage of the unit cost of our model of computation (e.g., a comparison of two strings packed into a word of

[1] This theorem is also cited in [3], where a different construction can be found.

O(log n) bits requires 1 unit of time). Here $prefix_l(z)$ ($suffix_l(z)$) denotes the prefix (suffix) of the string z of length l. The pseudo-code given below gives a detailed account of the preprocessing:

```
Procedure 2.1
begin
  $m \leftarrow \lceil n/\log n \rceil$;
  forall $1 \leqslant i \leqslant m$ pardo
    processor $p_i$ computes the lsp $s_i$ of $x_i$;
    comment Use one of the linear algorithms given in [6, 7, 9, 11].
    $w_i \leftarrow s_i \ldots s_{(i+3) \bmod m}$;
    comment The string $w_i$ is defined to be the substring of the circular string $x$ that
    starts at the same position as $s_i$ (denoted as $pos(s_i)$) and terminates at
    $pos(s_{(i+3) \bmod m})$.
    processor $p_i$ is assigned to the string $w_i$;
    processor $p_i$ "packs" $prefix_l(w_i)$, $suffix_r(w_i)$, $1 \leqslant l \leqslant |w_i|$, $1 \leqslant r \leqslant |z_i|$;
    comment The number of prefixes (suffixes) of $w_i$ is O(log $n$). The processor can
    sequentially pack all of them into single words (of length at most O(log $n$) each) in
    O(log $n$) units of time.
  odpar
end
```

It is not difficult to see that the input string can be written as

$$x = suffix_{l_1}(w_{m-2})prefix_{l_2}(w_1)prefix_{l_2}(w_2)\ldots prefix_{l_m}(w_m) \text{ with}$$

$$m = \lceil n/\log n \rceil, \tag{2.1}$$

where $l_1 = n - pos(s_m) + pos(s_1)$, $l_i = pos(s_i) - pos(s_{i-1})$, $2 \leqslant i \leqslant m$. The following theorem can be directly derived from the above remarks.

Theorem 2.2. *Procedure* 2.1 *packs all prefixes and suffixes of the substrings defined by the lsp's in* O(log n) *time using* O(n/log n) *processors and linear space.*

3. The duel of two local starting points

Let s_i and s_j be local starting points of the blocks x_i and x_j, respectively and w.l.o.g. let $j > i$. We are in pursuit of evidence that one of the two lsp's is not the starting point of an llr (there is of course the case that both can be starting points of an llr, in which case we keep the leftmost one). Lemma 1.1 provides the criterion that will aid us in eliminating one of the indices i or j; it suffices to compare the strings

$$w = s_i \ldots s_{i+1} \ldots s_j \quad \text{and} \quad z = s_j \ldots s_{j+1} \ldots s_{2j-i},$$

where w is the substring of the input string x starting at the $pos(s_i)$ and z is the substring of x starting at $pos(s_j)$, both of length $k+1$. The comparison of w and z can be performed by splitting them into substrings, which we already have in the library of "packed" strings computed at the preprocessing stage. Formally, the splitting can be done as follows: Let $C=\{c_r,\ i\leqslant r\leqslant j\}$, where c_r denotes the length of the string $s_i \ldots s_r$ (prefix of w) (here $c_i=1$). Similarly, let $D=\{d_q,\ j\leqslant q\leqslant 2j-i\}$, where d_q denotes the length of the string $s_j \ldots s_q$ (prefix of z) (here also $d_j=1$). Moreover, let $A=\alpha_1,\ldots,\alpha_{2(j-i)}$ denote the ascending sequence of the elements of the set $C\cup D$. Elements of A which come from C (D) we refer to as c (d) elements. Then one can easily prove the following lemma.

Lemma 3.1. *Any four consecutive elements of the A sequence include at least one member of C and at least one member of D.*

Proof. Assume that α_ρ, $\alpha_{\rho+1}$, $\alpha_{\rho+2}$, $\alpha_{\rho+3}$ are all members of C—say c_τ, $c_{\tau+1}$, $c_{\tau+2}$, $c_{\tau+3}$. Let α_λ, for some $0<\lambda<\rho$ and α_μ, for some $0>\mu>\rho+3$ be such that

$$\alpha_\lambda\in D \quad \text{and} \quad \alpha_{\lambda+v}\notin D \quad \text{for all } 0<v<\rho-\lambda$$

and

$$\alpha_\mu\in D \quad \text{and} \quad \alpha_{\mu-v}\notin D \quad \text{for all } 0<v<\mu-\rho-3.$$

Moreover, let $\alpha_\lambda=d_l$ and $\alpha_\mu=d_{l+1}$. (This follows from the fact that α_λ is a member of D, say d_l, and d_l is the nearest "d" to the left of c_τ in A and d_{l+1} is the nearest "d" to the right of $c_{\tau+3}$ in A and there are no other "d"'s between them.) From the fact that $\alpha_\lambda<\alpha_\rho<\alpha_{\rho+3}<\alpha_\mu$ it follows that

$$\alpha_\mu-\alpha_\lambda>\alpha_{\rho+3}-\alpha_\rho. \tag{3.1}$$

Moreover, the string $s_i\ldots s_{\tau+3}$ (its length is $\alpha_{\rho+3}$) has the blocks $x_{\tau+1}$ and $x_{\tau+2}$ as proper substrings but that is not the case for the string $s_i\ldots s_\tau$ (its length is α_ρ). Thus,

$$\alpha_{\rho+3}-\alpha_\rho>|x_{\tau+2}|+|x_{\tau+1}|=2\lceil\log n\rceil. \tag{3.2}$$

Inequalities (3.1) and (3.2) imply that

$$d_{l+1}-d_l=\alpha_\mu-\alpha_\lambda>2\lceil\log n\rceil. \tag{3.3}$$

Inequality (3.3) implies that the distance of s_l and s_{l+1} is greater than the size of two blocks, a contradiction since s_l and s_{l+1} belong to consecutive blocks.

The case of four consecutive elements being members of D is shown similarly. □

Let B be the sequence formed from A by removing all but the first c of every subsequence of consecutive c's and all but the first d of every subsequence of consecutive d's. Now consider splitting the strings w and z into the substrings defined

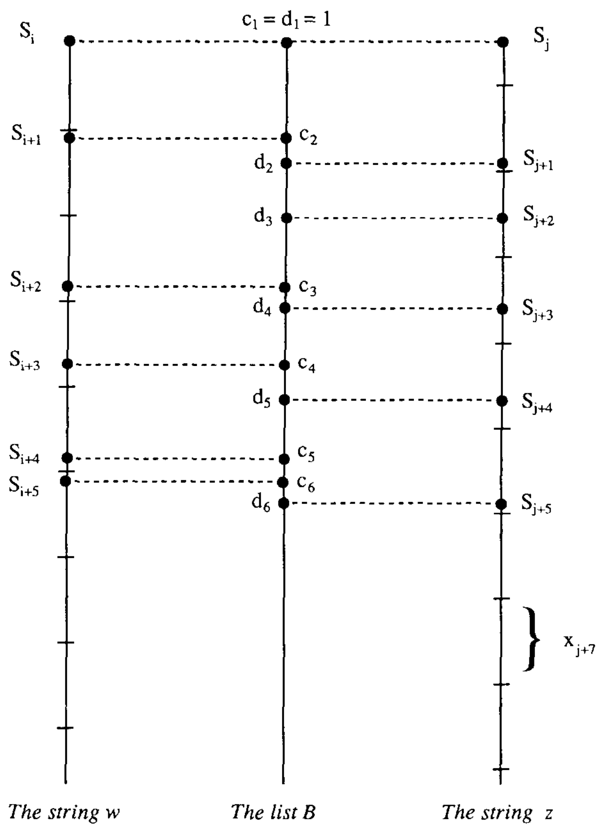

Fig. 1.

by the positions of the B sequence. One can see that all such substrings can be found "packed" among the prefixes and suffixes in the preprocessing stage (see Fig. 1). A detailed account of the duel between two adjacent strings is given below in pseudo-code:

```
Procedure 3.2 DUEL ($w$, $z$)
begin
  forall $i \leqslant m \leqslant 2j-1$ pardo
    Processor $p_m$ is assigned to the point $s_m$;
    if $i \leqslant m \leqslant j$ then
      Processor $p_m$ writes $c_m = |s_i \dots s_m|$ into a doubly linked list $C$;
    else
      Processor $p_m$ writes the position $d_m = |s_j \dots s_m|$ into a doubly linked list $D$;
    Processor $p_m$ computes its position in the doubly linked list A (as defined above);
    comment Processor $p_r$, $i \leqslant r \leqslant j$ inserts $c_\tau$ into the $D$ list; it is not difficult to see that
    $c_\tau$ has to be inserted in one of the positions between $d_{r-2}$ and $d_{r+2}$ and, thus, that
    it is sufficient for processor $p_r$ to check whether $c_{r-1}$ or $c_{r+1}$ must be linked
    with $c_r$.
    Then all processors can calculate their ranking in the list A (see Fig. 1).
    Let $A = \{\alpha_1, \ldots, \alpha_{2(j-i)}\}$ and let processor $p_m$ be attached to $\alpha_m$;
    if $\alpha_m$ and $\alpha_{m-1}$ are both in $C$ or $D$ then
      Processor $p_m$ deletes $\alpha_m$ and marks itself idle;
      comment The new list is the doubly linked list $B$ defined above.
    if $(w_{\alpha_m}, w_{next(\alpha_m)}) \neq (z_{\alpha_m}, z_{next(\alpha_m)})$ then                    (3.4)
      comment $(w_{\alpha_m}, w_{next(\alpha_m)})$ is the substring of $w$ that starts at position $\alpha_m$ and
      terminates at position $next(\alpha_m)$ (that is the adjacent element of $\alpha_m$ in $B$).
      The string $(z_{\alpha_m}, z_{next(\alpha_m)})$ is defined similarly.
      Processor $p_m$ is marked "winner";
      comment All comparisons can be done in constant time by processor $p_m$ since
      all strings involved have been packed into single words at the preprocessing
      stage.
    if no processor is marked "winner" then return $w$;
    Compute the smallest $v$ such that $p_v$ is marked winner;                                      (3.5)
    comment This can be done in constant time, see Lemma 3.3.
    if $(w_{\alpha_v}, w_{next(\alpha_v)}) \leqslant (z_{\alpha_v}, z_{next(\alpha_v)})$ then                    (3.6)
      Processor $p_v$ returns $w$
    else
      Processor $p_v$ returns $z$
  odpar
end
```

Lemma 3.3 (Galil [8]). *The computation of the smallest indexed processor at step* (3.5) *above can be done in constant time using* $O(j-i)$ *processors.*

The procedure DUEL makes use of the "library" of strings computed at the preprocessing stage; all steps require constant time and at any time DUEL uses at most $2(j-i)$ processors. Thus, we have the following lemma.

Lemma 3.4. *Procedure* (3.2) *requires* $O(1)$ *units of time and* $O(j-i)$ *processors for executing the duel between* w *and* z.

4. Computing the llr over fixed alphabets

The main algorithm for computing the canonical structure of a circular string falls within the following framework. First preprocessing reduces the problem from size $O(n)$ to one of size $O(n/\log n)$. From (2.1) and Procedure 2.1 one can see that the input string can be expressed as a string of length $O(n/\log n)$ over the alphabet defined by the set $\Sigma' = \{prefix_r(w_i), suffix_r(w_i), 1 \leqslant r \leqslant \lceil \log n \rceil$ and $1 \leqslant i \leqslant \lceil n/\log n \rceil\}$, with w_i as in Procedure 2.1. Thus, the size of the problem has been reduced to $O(n/\log n)$.

All symbols (over Σ' as in (2.1)) of the input string x are paired with a processor and they are attached as leaves to a full binary tree (w.l.o.g. we assume that the number of symbols is a power of 2). Initially, every symbol is considered to be a candidate for the starting point of the llr. Every pair of siblings duels for the occupation of the parent node. The duel is executed by means of Procedure 3.2. The winner moves to the parent node and prepares for a new duel with its new sibling; the loser teams up with the winner for the new duel. The index of the processor that conquers the root is the required starting point of the least lexicographic rotation of the input string.

Initially, $O(n/\log n)$ processors are attached to the leaves. Each duel at the ith level involves strings of length 2^i and there are as many processors available to perform it; this can be done in constant time by means of Procedure 3.2 (see Lemma 3.4). Since the height of the tree is at most $\log n$, the overall time required is $O(\log n)$ and no more than $O(n/\log n)$ processors are needed at any time; this together with Theorem 2.2 leads us to the proof of Theorem 1.1. The correctness of the algorithm is a direct application of Lemma 1.1.

A simpler version of the algorithm given above is possible (e.g., [3]), which bypasses the rather lengthy procedure DUEL (improving the running time by a constant factor). Here we present this version due to its direct adaptation to general alphabets, which is the main goal of this paper; thus, we avoid unnecessary duplication of the results.

5. Preprocessing over general alphabets

The preprocessing procedure of Section 2, when applied to a string over a general alphabet (of size $|\Sigma|$) will not necessarily lead to a space reduction, since the packing into single words will take $O(\log n \log |\Sigma|)$ bits of space and, therefore, a comparison of two prefixes or suffixes would cost $O(\log |\Sigma|)$.

One can see from the main algorithm of Section 4 that it suffices to have efficient procedures which execute the DUEL; in particular, we draw attention to the following:

(i) In order to execute the string comparison at step (3.4), we only need evidence that the two strings involved are different.

(ii) At step (3.6) we compare two strings of length $O(\log n)$.

Both (3.4) and (3.6) require constant time in the case of fixed alphabets. Here we shall employ data structures for all prefixes and suffixes—as defined in Section 2—that allow the issue of certificates of difference in constant time, except in the case of comparing a suffix versus a prefix where the cost is $O(\log \log n)$ units of time. The overall preprocessing for unbounded alphabets is subdivided into the following tasks:

(1) We merge all prefixes into a data structure (defined below) called a merged prefix tree, that will provide answers for comparisons between prefixes.

(2) We merge all suffixes into a merged suffix tree, that will provide answers for comparisons between suffixes.

(3) We check all prefixes for membership in the merged suffix tree.

5.1. Computing the merged prefix tree

Given a set of strings $\{w_i\}$, $1 \leqslant i \leqslant n/\log n$ (as they are defined in Procedure 2.1), we will construct a data structure—the merged prefix tree—that will contain all information about all the prefixes of w_i, $1 \leqslant i \leqslant n/\log n$. The *merged prefix tree* (or *prefix tree of a dictionary*) is defined as follows (see Fig. 2):

(i) The merged prefix tree is a rooted tree.

(ii) Each edge (x, y) is labeled with (j, w_i), the jth symbol of w_i for some $1 \leqslant i \leqslant n/\log n$, where j is the length of the path from the root to y, and x is the node closer to the root.

(iii) The concatenation of the symbols represented by the labels on the edges on a path from the root to a node is equal to $prefix_j(w_i)$, where j is the length of the path.

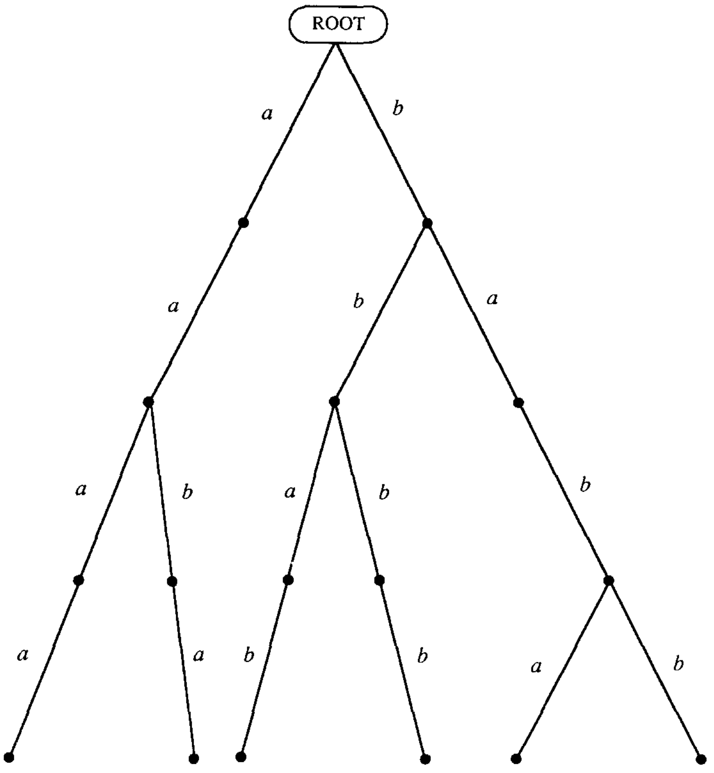

Fig. 2. The merged prefix tree for the dictionary $d = \{aaaa, baba, aaba, babb, bbab, bbbb\}$.

(iv) No two sibling edges may have the same label.
(v) Every prefix of every w_i appears in the tree.

The construction of the merged prefix tree is done by assigning a processor to every w_i, which starts building the merged prefix tree from the root towards the leaves and labeling the edges sequentially; the computation of the different labels of the edges descending from a node at each level is done my means of concurrent writes to Bulletin Boards (*BB* or *NBB*) held in the common memory. The pseudo-code below gives a detailed account of the method:

```
Procedure 5.1
begin
  Processor $p_i$, $1 \leqslant i \leqslant n/\log n$, is assigned to the string $w_i$;
  for $j = 1$ to $\log n$ do
    Let BB be an $1 \times |\Sigma|$ matrix and let $(j, w_i)$ denote the $j$th symbol of $w_i$;
    Processor $p_i$, $\forall 1 \leqslant i \leqslant n/\log n$, writes $i$ in $BB((j, w_i))$;
    Processor $p_i$, $\forall 1 \leqslant i \leqslant n/\log n$, stores $ID(j, w_i) \leftarrow BB((j, w_i))$ in its own memory;
    comment Here we assign a unique (to all $j$th symbols) integer from the set of
    processor indices. This is done in order to reduce the domain (the range of
    symbols is $|\Sigma|$ and the range of indices in $O(n/\log n)$) of the Bulletin Board NBB
    at the $j$th level (see below).
  od
  All processors are attached to the root (marked "fresh");
  for $j = 1$ to $\log n$ do
    forall processors attached to node marked "fresh" pardo
    An $1 \times n/\log n$ bulletin board NBB is attached to each node marked "fresh";
    comment The matrix NBB is held in the common memory and its domain is the
    set of processor indices.
    Processor $p_i$ writes its index $i$ in $NBB(ID((j, w_i)))$;
    Processor $p_i$ with $i \neq NBB(ID((j, w_i)))$ is marked "loser";
    Processor $p_{NBB(ID((j, w_i)))}$ is marked as "winner";
    comment The "winners" are processors handling distinct symbols. A loser
    handles a symbol that is already handled by a winner.
    The "winner" creates a new node as child of its associated node marked "fresh";
    The "winner" marks the new node as "fresh" and its parent "old";
    comment The new node is marked for tree expansion at the next iteration.
    The "winner" labels the new edge with $(j, w_i)$;
    The "winner" gets attached to the "fresh" node;
    The "losers" get attached to the "fresh" node created by the "winner"
    processor $p_{NBB(ID((j, w_i)))}$;
    odpar
  od
end
```

In fact, in Procedure 5.1 we need to name the nodes created by the processors; a node is named (j, d) if it is created by processor p_j at depth d (note that $j = O(n/\log n)$ and $d = (O(\log n))$. Also, whenever a processor creates a new node, say *node*, it creates a labeled edge (with label l) connecting it with the *parentnode*; let $LINK(parent$-$node, l) = node$ denote the connection. The list *LINK* is used without initialization since its size is $O(n^2)$ and also it might contain false information. But whenever a processor creates a new node—say *node*, it updates the list *VALIDITY* $(node) = (parentnode, label, node)$, where *label* is the label of the edge connecting *parentnode* and *node*. The list *VALIDITY* has been initialized to empty in $O(\log n)$ units of time using $O(n/\log n)$ processors and the updating of *VALIDITY* takes constant time. Also the size of *VALIDITY* is $O(n)$. This will allow us to answer queries such as:

Given a node, say n, and a symbol l, we want to find whether there exists a child c such that the edge (n, c) is labeled with l. A child c is given by $LINK(n, l)$, but it might not be a true one since *LINK* has not been initialized. Then by checking the contents of $VALIDITY(LINK(n, l))$ one can decide whether $LINK(n, l)$ is a valid one.

Lemma 5.2. *The procedure above correctly constructs the merged prefix tree in* $O(\log n)$ *units of time, using* $O(n/\log n)$ *processors and* $O(n^2/\log^2 n + |\Sigma|)$ *space.*

Proof. At the initialization stage we assign integers to each symbol. It is possible that two different symbols are assigned the same integer; one can see that $ID(j, w_i) = ID(j', w_{i'})$ only for some $j \neq j'$, since a processor assigns its index only once at each iteration. This will not cause problems in the identification of different labels assigned via concurrent writes in NBB since, for each j, different symbols have different IDs and identical symbols have the same ID. In other words

$$ID(j, w_i) = ID(j, w_{i'}) \iff (j, w_i) = (j, w_{i'}).$$

Otherwise, the construction of the merged suffix tree is a straightforward application of the definition, together with the "competitions" on the Bulletin Board *NBB* in order to establish the different labels descending from each node.

The time analysis follows immediately. The bulletin board BB requies $O(|\Sigma|)$ units of memory and, furthermore, at each iteration we require at most $O(n/\log n)$ Bulletin Boards, one for each node marked "fresh"; the size of each NBB is bounded by $O(n/\log n)$. □

5.2. Computing the merged suffix tree

Here we compute a data structure T_d on the set of strings $d = \{w_1, \ldots, w_{n/\log n}\}$, called the merged suffix tree. We assume, w.l.o.g. that each string of d terminates with a special symbol \$ which is not in the alphabet and which is used as an endmarker. T_d is defined as follows (see Fig. 3):

(i) The merged suffix tree is a rooted tree with labeled edges.

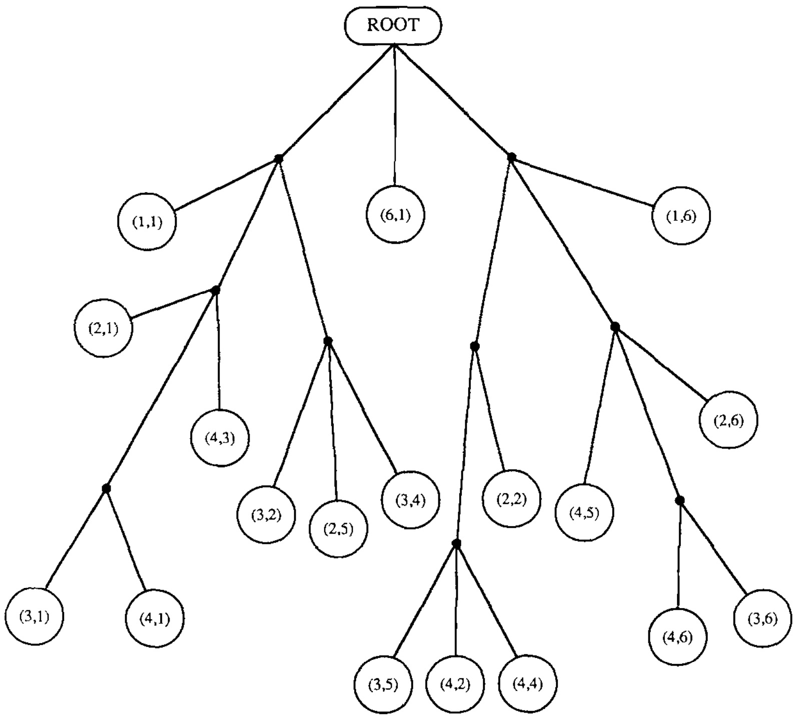

Fig. 3. The merged suffix tree T_d for the dictionary $d=\{aaaa, baba, aaba, babb, bbab, bbbb\}$. If two or more suffixes are equal, only one is represented on the tree. Here we use strings as labels for exposition purposes only. Also a leaf is marked (s, t), representing the suffix starting at the sth position of the tth word of the dictionary.

(ii) Each edge is labeled with an identifier $ID(w_i, i_j, l)$, which represents the substring of w_i that starts at position i_j and has length l (this is a prefix of the suffix of w_i that starts at position i_j).

(iii) No two sibling edges have the same (nonempty) prefix.

(iv) Each leaf is labeled with $[i_j, w_i]$, where i_j is a distinct position of w_i.

(v) The concatenation of the strings represented by the labels of the edges on a path from the root to leaf $[i_j, w_i]$ equals the suffix of w_i starting at position i_j.

The construction of the merged suffix tree is done in two stages:

(1) We construct an approximate version of the suffix tree of w_i for all $1 \leqslant i \leqslant \lceil n/\log n \rceil$—this is called the merged skeleton tree.

(2) We refine the merged skeleton tree where necessary. Here processor allocation plays a key role in the construction.

5.2.1. *The merged skeleton tree*

An approximate version of the suffix tree is computed first. The merged skeleton tree S_d with $d=\{w_1, \ldots, w_{n/\log n}\}$ (see Fig. 4) is defined as follows:

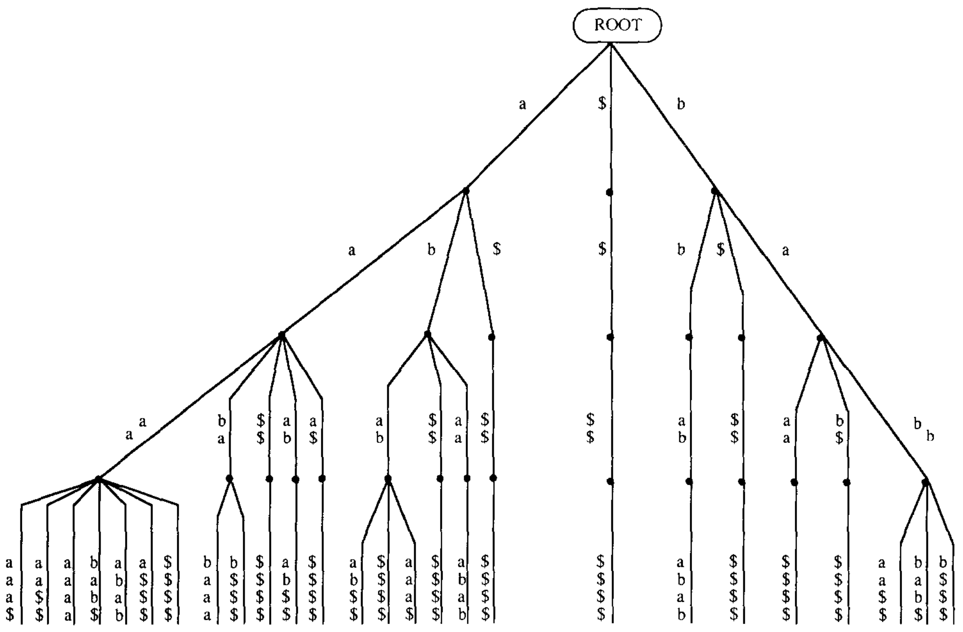

Fig. 4. The merged skeleton tree S_d for the dictionary $d = \{aababaaa, aaaaabab, aaaaaaaa, bbababab\}$. Here we use strings as labels for exposition purposes only.

(i) It is a rooted tree.

(ii) Each edge is labeled with an identifier $ID(w_i, i_j, 2^l)$, where l is the length of the path from the root to the node (nearest to the root) attached to the edge; the identifier represents the substring of w_i that starts a position i_j and has length 2^l.

(iii) No two sibling edges may have identical labels.

(iv) Each leaf is labeled with (i_j, w_i), where i_j is a distinct position of the string w_i.

(v) The concatenation of the strings represented by the labels of the edges on a path from the root to leaf (i_j, w_i) is the suffix of w_i starting at position i_j.

Processors handling suffixes match identical prefixes via their IDs which are uniquely defined by virtue of the write conflict rule of CRCW PRAMs. Furthermore, the ID of a long string is initially combined from the pair of IDs of the two substrings created when we split the string in half; then using the concurrent write conflict mechanism, we replace the pair with a new unique value. (This is done in order to keep the length of the IDs constant.) The method is outlined as follows:

```
Procedure 5.3
begin
    Let BB be a $1 \times |\Sigma|$ Bulletin Board;
    Processor $p_{ij}$ is associated with the $j$th symbol of $w_i$, $1 \leqslant i \leqslant n/\log n$, $1 \leqslant j \leqslant \log n$
    forall $i, j$ pardo
    processor $p_{ij}$ writes its index in $BB((j, w_i))$;
    $ID(w_i, j, 1) \leftarrow BB((j, w_i))$;
    comment The creation of each ID is followed by the creation of a new node
    linked with the root as a child; the linking edge is labeled with the ID.
    for $k = 1$ to $\lceil \log\log n \rceil$ do
      Let NBB be an $n \times n$ Bulletin Board.
      processor $p_{ij}$ writes its index in $NBB(ID(w_i, i_j, 2^{k-1}), ID(w_i, i_j + 2^{k-1}, 2^{k-1}))$;
      $ID(w_i, j, 2^k) \leftarrow NBB(ID(w_i, j, 2^{k-1}), ID(w_i, j + 2^{k-1}, 2^{k-1}))$;
      comment The above statement is executed by processor $p_{ij}$; the same processor
      creates a new node and links it to the lower node of the edge labeled with
      $ID(w_i, j, 2^{k-1})$. The new edge is labeled with $ID(w_i, j, 2^k)$.
    od
  odpar
end
```

Theorem 5.4. *Procedure* 5.3 *correctly computes the merged skeleton tree in* $O(\log\log n)$ *units of time using* $O(n)$ *processors and* $O(n^2 + |\Sigma|)$ *space.*

Proof. The construction is a straightforward application of the definition. The initialization requires at most $O(1)$ units of time and $O(n)$ processors. Furthermore, the number of repetitions is $O(\log\log n)$, each of them costing $O(1)$ units of time. □

Corollary 5.5. *The computation of the merged skeleton tree can be done in* $O(\log n \log\log n)$ *units of time using* $O(n/\log n)$ *processors.*

5.2.3. Refining the merged skeleton tree

The refinement of the merged suffix tree will be done in two stages. First the processors are distributed on the merged skeleton tree, and then the set of processors attached to the children of a node perform a local refinement. We will see that these refinements suffice to refine the whole tree.

The processor allocation method is called *the migration of orphans*. We use n processors, all initially attached to the leaves of the merged skeleton tree (there are exactly n leaves as many as suffixes). Now the processors "compete" for the occupation of the parent node (of the set of siblings that they are attached to); only one processor succeeds and moves to the parent node. The competition is repeated in the next level up until the root is reached. The result is that every set of k sibling nodes has $k-1$ processors attached to them; the one child without a processor is called an *orphan*. The $k-1$ processors attached to the children of every node "elect" one processor (by concurrently writing in the same common memory location) to simulate the orphan node; this will introduce a factor of 2 in the time complexity, but the processor allocation is simple and recursive calls can be done during the refinement process (see Procedure 5.6) without major processor reallocation.

The merged skeleton tree needs to be refined in every node (except the root) that has two or more children; the labels of their outgoing edges are different but they might have a common prefix. The refinement at each node can be done independently; therefore, we are presenting the refinement of just a portion of the merged skeleton tree consisting of a node and its children, i.e., *a local refinement.* The local refinement is done by means of binary search. Let n_z be a node of the merged skeleton tree and let c_j, $1 \leqslant j \leqslant m$ be the children of n_z. Moreover, let z_j, $1 \leqslant j \leqslant m$, be the set of strings (all of the same length) that are represented by the labels of the edges (n_z, c_j), $1 \leqslant j \leqslant m$. All but one (say c_m—orphan) nodes have one processor attached to them—a processor allocation produced by the migration of the orphans. Let $ID(w_{i_j}, q_j, |z_j|)$ be the already computed labels of these edges. The pseudo-code below outlines the method for refining this part of the tree.

```
Procedure 5.6
begin
  forall $1 \leqslant j \leqslant m-1$ pardo
    $l_j \leftarrow |z_j|$; $n_j \leftarrow n_z$;
    comment Node $n_j$ is used as a reference point of the processor allocation.
    Processor $p_j$ writes its index in $AUX(n_j)$;
    comment $AUX$ is a common memory array.
    Processor $p_{AUX(n_j)}$ is elected to simulate the "missing" processor $p_m$;
    comment In the sequel we assume the existence of $p_m$ for exposition purposes.
    while $l_j \neq 1$ do
      $l_j \leftarrow l_j/2$;
      processor $p_j$ writes $j$ in $BB(ID(w_{i_j}, q_j, l_j), n_j)$;
      comment This competition establishes whether there are any common prefixes
      of size $l_j$ among the strings represented by the labels.
      if $p_j$ is a "loser" $(j \neq BB(ID(w_{i_j}, q_j, l_j), n_j))$ then
        Processor $p_j$ writes $j$ in $AUX(BB(ID(w_{i_j}, q_j, l_j), n_j))$;
        $l_j \leftarrow AUX(BB(ID(w_{i_j}, q_j, l_j), n_j))$;
        comment Processor $p_j$ is a "loser"; this implies that at least another edge has
        a label with the same ID; $p_{l_j}$ is elected as the leader of all the losers with the
        same ID. Note that several leaders may be elected concurrently.
        $n_j \leftarrow n_{AUX(BB(ID(w_{i_j}, q_j, l_j), n_j))}$;
        $q_j \leftarrow q_j + l_j$;
        comment Processor $p_{l_j}$ creates a new node $n_j$ as the child of
        $n_{BB(ID(w_{i_j}, q_j, l_j), n_j)}$ and labels the edge between $n_j$ and $c_j$ with $ID(w_{i_j}, q_j, l_j)$.
        All edges with the same common prefix of length $l_j$ have been replaced by
        one edge connecting the original parent node (of $c_j$) and the new node
        $n_j$ (named by the leader of the "losers").
        Processor $p_{l_j}$ creates a new node $c'_{l_j}$ as a child of $n_j$;
        Processor $p_{l_j}$ labels the new edge with $ID(w_{i_j}, q_j, l_j)$;
```

```
            comment The node $c_{l_j}$ is going to be an orphan at the next iteration. Now the
            leader and its associated "losers" have the task of refining the labels of all the
            edges descending from $n_j$.
        else
            comment Processor $p_j$ is a "winner" if it succeeds in writing its index in BB.
            $c_j \leftarrow n_{AUX(BB(ID(w_{i_j}, q_j, l_j), n_j))}$;
            Processor $p_j$ labels the edge $(n_j, c_j)$ with $ID(w_{i_j}, q_j, l_j)$;
            comment Now all "winners" have the task of refining all the labels of the
            edges descending from $n_j$. The node $c_m$ is an orphan node.
    od
  odpar
end
```

Theorem 5.7. *Procedure* 5.6 *correctly refines the input subtree of the merged skeleton tree in* $O(\log \log n)$ *units of time and uses* $O(m)$ *processors.*

Proof. The refinement method is a straightforward parallel application of binary search. The processors compute all labels representing strings with the same common prefix of a certain length (using concurrent writes in the common memory); then they replace these edges with one edge labeled with the common ID. They repeat the procedure searching for common prefixes of half the length until the tree is fully refined; in fact, the next iteration is a new set of local refinements involving labels representing strings of half the length.

Moreover, one can see from the comments of Procedure 5.6 that the "orphan rule" is preserved from one iteration to the next one, when the original problem has been split up into two or more subproblems.

The length of the strings represented by the labels is at most $O(\log n)$ and performing a binary search on them takes $O(\log \log n)$ units of time. □

Using the migration of orphans we can allocate the processors on the merged skeleton tree in $O(\log n)$ units of time and by using the procedure given above we can refine it in $O(\log \log n)$ units of time using n processors, which implies the following corollary.

Corollary 5.8. *The construction of the merged suffix tree can be done in* $O(\log n \log \log n)$ *unis of time using* $O(n/\log n)$ *processors.*

5.3. Membership in the merged suffix tree

We shall make use of a list (called INFO) that will enable us to check all prefixes of a string w_i for membership in the merged suffix tree; the output will be a table containing all such memberships.

It is not difficult to modify the construction procedure of the merged suffix tree in order to accommodate the list *INFO* at every internal node. The variable *INFO*(*node*, *symbol*) is assigned the value (*nextnode*, *string*), where *node* is the node to which the list relates, *symbol* is any given symbol of the alphabet and the edge between *node* and *nextnode* is labeled with a label that represents a *string* whose first symbol is *symbol*. The list *INFO* is used without initialization but one can validate its contents by using a function similar to *VALIDITY* used in the case of merged prefix tree.

The procedure below makes use of only one processor which checks $prefix_j(w_i)$, $1 \leqslant j \leqslant |w_i|$, for membership in T_d sequentially (testing for membership of $prefix_{j+1}(w_i)$ is done after checking $prefix_j(w_i)$ and involves only the $(j+1)$th symbol). Testing is done by following the path from the root to the leaves and simultaneously matching the symbols of the prefixes with the strings represented by the labels in T_d; whenever a choice is possible (nodes with two or more children) the processor uses the list INFO which guides the choices of descent. The endmarker \$ (strings represented by labels on leaf edges in T_d, as shown in Fig. 3) signals a successful match. This will allow efficient testing for membership as follows:

```
Procedure CHECK ($w_i$, $T_d$)
begin
  $t \leftarrow 1$;
  $(NODE, STRING) \leftarrow INFO(ROOT, (j, w_i))$
  for $j = 1$ to $\lceil \log n \rceil$ do
    if $(t, STRING) = \$$ then
      return $prefix_{j-1}(w_i) = suffix_{j-1}(NODE)$;
      comment Here $NODE$ is a leaf of $T_d$.
    else
      if $(t, STRING) \neq (j, w_i)$ then exit;
    if $|STRING| = t$ then
      if $INFO(NODE, \$) \neq \emptyset$ then $prefix_j(w_i) = suffix_j(LEAF)$;
      comment The $LEAF$ is defined by $INFO(NODE, \$)$.
      $(NODE, STRING) \leftarrow INFO(NODE, (j, w_i))$;
      $t \leftarrow 1$;
    $t \leftarrow t + 1$;
  od
end
```

It is not difficult to show the following lemma.

Lemma 5.9. *Procedure CHECK correctly checks all prefixes of w_i for membership in T_d in* O(log *n*) *units of time.*

Theorem 5.10. *One can check all prefixes of the dictionary $\{w_1, \ldots, w_{\lceil n/\log n \rceil}\}$ in* O(log *n*) *units of time using* O(*n*/log *n*) *processors.*

6. Duel of strings and computing the llr over general alphabets

The procedure DUEL for fixed alphabets will be used for unbounded alphabets with two modifications for the actions taken at (3.4) and at (3.6).

At step (3.4) of the procedure DUEL we need to know whether two prefixes (suffixes) are different. If both are suffixes then the merged suffix tree provides the answer in constant time; if both are prefixes then the merged prefix tree provides the answer, and if one is a suffix and the other is a prefix then the table constructed by procedure CHECK can provide the answer also in constant time. Thus, the time requirements of step (3.4) remain the same as in Theorem 3.4 using the information computed at the preprocessing stage.

At step (3.6) of procedure DUEL we have to compare two prefixes (suffixes) and one processor is provided. If both are prefixes (suffixes), then one can compare them in $O(\log\log n)$ units of time using the merged prefix tree (merged suffix tree) by computing their common ancestor. If one of the strings is a suffix (say $suffix_l(w_k)$) and the other is a prefix (say $prefix_l(w_i)$), then the comparison is done as follows:

(i) It is not difficult to modify procedure CHECK in order to provide the following function:

$$e(prefix_l(w_i)) = \begin{cases} j & \text{if procedure } CHECK(w_i, T_d) \text{ exited at level } j, \\ l & \text{otherwise.} \end{cases}$$

This function will enable us to calculate the node, say n_w, of the skeleton tree such that the string represented by the label of the path from the root to n_w equals the longest common prefix $prefix_l(w_i)$, whose length is a power of two.

(ii) Now we compute the common ancestor of the n_w and the parent node of the leaf of the skeleton tree that represents the $suffix_l(w_k)$ tree. If the suffix node is different from the prefix node then they both "move" to their parent node on the skeleton tree. This step is repeated until they are in the same node (common ancestor), say n_a. Let n_p and n_s be the nodes visited by the prefix and suffix at the previous step; both are children of n_a. This computation requires in the worst case $O(\log\log n)$ units of time—the height of the tree.

(iii) When the common ancestor is found, the comparison of $prefix_l(w_i)$ and $suffix_l(w_k)$ is reduced to comparing the strings represented by the labels of the edges between n_a and the two children n_p and n_s; their length is at most $O(\log\log n)$. Therefore, one processor can complete the comparison in $O(\log\log n)$ units of time.

The time requirement of the step above is $O(\log\log n)$ units of time, which is also the bottleneck of the procedure DUEL for unbounded alphabets. This implies that the main algorithm of Section 4—which remains the same for the case of unbounded alphabets—has its time complexity increased by a factor of $O(\log\log n)$ and, thus, leads us to Theorem 1.3.

Note. Recently, Apostolico and Crochemore [2] also gave a CRCW PRAM algorithm for Lyndon factorization with applications to the canonization of circular strings; it requires $O(\log n)$ units of time using n processors.

References

[1] A.G. Akl and G.T. Toussaint, An improved algorithmic check for polygon similarity, *Inform. Process. Lett.* **7** (3) (1978) 127–128.

[2] A. Apostolico and M. Crochemore, Fast parallel Lyndon factorization with applications, CSD-TR-931, Purdue University, 1989; revised 1990.

[3] A. Apostolico and C. Iliopoulos, Unpublished manuscript.

[4] A. Apostolico, C. Iliopoulos, G.M. Landau, B. Schieber, and U. Vishkin, Parallel construction of a suffix tree with applications, *Acta Algorithmica* **3** (1988) 347–365.

[5] A. Apostolico, C.S. Iliopoulos and R. Paige, An $O(n \log n)$ cost parallel algorithm for the one function partitioning problem, in: Jung and Mehlhorn, eds, *Proc. Int. Conf. on Parallel architectures*, (Akademie-Verlag, Berlin, GDR, 1987) 70–76.

[6] K.S. Booth, Lexicographically least circular substrings, *Inform. Process. Lett.* **10** (4) (1980) 240–242.

[7] J.P. Duval, Factorizing words over an ordered alphabet, *J. Algorithms* **4** (1983) 363–381.

[8] Z. Galil, Optimal parallel algorithms for string matching, in: *Proc. 16th ACM symp. on Theory of Computing* (1984) 240–248.

[9] C.S. Iliopoulos and W.F. Smyth, A new sequential algorithm for canonization of circular strings, submitted.

[10] D. Knuth, J. Morris and V. Pratt, Fast pattern matching in strings, *SIAM J. Comput.* **6** (2) (1977) 323–350.

[11] Y. Shiloach, Fast canonization of circular strings, *J. Algorithms* **2** (1981) 107–121.