ELSEVIER

Information Processing Letters 50 (1994) 239–246

Information Processing Letters

An optimal algorithm to compute all the covers of a string

Dennis Moore [a], W.F. Smyth [a,b,*]

[a] *School of Computing, Curtin University of Technology, Bentley, WA 6102, Australia*
[b] *Department of Computer Science and Systems, McMaster University, Hamilton, Ontario, Canada L8S 4K1*

Communicated by M.J. Atallah; received 8 July 1993; revised 23 February 1994

Abstract

Let x denote a given nonempty string of length $n = |x| \geqslant 1$. A string u is a *cover* of x if and only if every position of x lies within an occurrence of u within x. Thus x is always a cover of itself. In this paper we characterize all the covers of x in terms of an easily computed normal form for x. The characterization theorem then gives rise to a simple recursive algorithm which computes all the covers of x in time $\Theta(n)$.

Key words: Algorithms; Analysis of algorithms; Design of algorithms; Combinatorial problems; String covers

1. Introduction

Let x denote a *string* of *length* $n = |x| \geqslant 0$; in particular, let ε denote the *empty* string of length zero. For any nonempty string x, let $k \geqslant 1$ be the greatest integer such that

$$x = (v'v)^k v' \qquad (1)$$

over all possible choices of the strings v' (possibly empty) and $v \neq \varepsilon$. Then let v^* be the maximum length string v' which satisfies (1). Thus

$$x = (v^*v)^k v^*. \qquad (2)$$

We call the right-hand side of (2) the *normal form* of x; in Section 3 we show how the normal form can be efficiently computed. We remark that a "reverse normal form" $x = w^*(ww^*)^k$ can equally well be defined, and that the results of this paper, or analogous ones, can equally well be derived from it. (For example, for $x = ababa$, we can write either $x = (ab)^2a$ or $x = a(ba)^2$.)

Suppose that in (1) $k > 1$ and that there exist strings u' and u with $|u'| > |v'|$ such that

$$x = (v'v)^k v' = (u'u)^k u'. \qquad (3)$$

Then x has two distinct overlapping primitive periods, an impossibility by an argument given in [11]. We conclude that for $k > 1$, there is exactly one string $v' = v^*$ which satisfies (1). However, to show that, for $k = 1$, v' is not necessarily equal to v^*, consider the example $x = aabaa$: we might find $v' = a$, $v = aba$ in (1), but then in (2) we would choose $v^* = aa$, $v = b$. If $v^* = \varepsilon$, then we say that x is either *primitive* (for $k = 1$) or a *repetition* (for $k > 1$); if $v^* \neq \varepsilon$, we say that x is either *weakly periodic*

* Corresponding author.
Email: bill@maccs.dcss.mcmaster.ca.

(for $k = 1$) or *strongly periodic* (for $k > 1$).

For all strings y and z such that $x = yz$, we say that y is a *prefix* and z a *suffix* of x; if both y and z are nonempty, then we say that y is a *proper prefix* and z a *proper suffix* of x. A string which is both a proper prefix and a proper suffix of x is called a *border* of x. Thus in (1), if $v' \neq \varepsilon$, then x has border v'; we see in fact that a nonempty string is primitive if and only if it has no border. If x has suffix z and y has prefix z, we can form the string $x\{z\}y$ by *overlapping* the suffix of x with the prefix of y. For example, if $x = aabaaba$ and $y = ababa$, then we can form the strings

$$x\{a\}y = aabaab\underline{a}baba,$$
$$x\{aba\}y = aabaa\underline{aba}ba,$$
$$y\{a\}x = abab\underline{a}abaaba,$$

where the underscore denotes the overlapped substrings. In particular, for arbitrary strings x and y, we can always form $x\{\varepsilon\}y = xy$. Further, for any string x, we can always form $x = x\{x\}x$, the complete overlap of x with itself; in the case that x has a border z, we can also form $x\{z\}x$. If a given string x can be written in the form

$$x = u\{z_1\}u\{z_2\}u \ldots u\{z_m\}u, \tag{4}$$

for strings $u \neq \varepsilon$ and $z_1, z_2, \ldots, z_m$, where $m \geqslant 1$, then we say that u *covers* (or is a *cover* of) x. If moreover $u \neq x$, u is called a *proper cover* of x. Thus for $u \neq x$, u is a border of x, and each string z_i, $i = 1, 2, \ldots, m$, is either empty or a border of u. Thus every string with a proper cover u necessarily has border u, an observation which yields the following:

Lemma 1. *No primitive string has a proper cover.* □

In (4), if every $z_i = \varepsilon$, $i = 1, 2, \ldots, m$, it follows that $x = u^{m+1}$ is a repetition, and so the cover u may or may not be primitive. However, it is easy to see that, if $u \neq x$ and some z_i is not empty, then u cannot be primitive.

In the next section we prove two less obvious properties of covers, results that are then used to establish a characterization theorem for covers. In Section 3, based on this characterization, we present a linear time algorithm which computes all the covers of a given string x; this algorithm is a simplified and corrected version of an algorithm that has already been published in preliminary form [12]. Section 4 briefly discusses open problems and future work.

2. A characterization of covers

We introduce the main result of this section with a lemma:

Lemma 2. *Let u be a proper cover of x, and let $z \neq u$ be a substring of x such that $|z| \leqslant |u|$. Then z is a cover of x if and only if z is a cover of u.*

Proof. Clearly if z covers u, and u covers x, then z covers x. Suppose therefore that both z and u cover x. Then z is a border of x and hence also, since $z \neq u$, a border of u; thus z must also cover u. □

When a longest proper cover u of x has already been found, this lemma reduces the problem of finding shorter covers of x to the problem of finding covers of u. In particular, consider the case $k = 1$, where $x = v^*vv^*$ and v^* is the border of maximum length of x. Then no proper cover of x can have length greater than v^*, so that, by Lemma 2, every proper cover of x is a cover of v^*. More generally, we find that we can characterize the covers of x as follows:

Theorem 3. *Each cover u of x is either*
(a) *$(v^*v)^jv^*$, $j = 1, 2, \ldots, k$, or*
(b) *(if $k > 1$) a proper cover of v^*vv^*, or*
(c) *(if $k = 1$) a cover of v^* which also covers v^*vv^*.*

In order to prove this theorem, we shall need a second, more technical, lemma (a special case of a result due to Cummings [7,8]):

Lemma 4. *If for a given string x and some (nonempty) string $u \neq x$, $x = u\{z\}u = yzy'$,* *then* $x = y^j y^*$, *where*

(a) $j = \lfloor |z|/|y| \rfloor + 2$;

(b) y^* *is a prefix of* y *of length* $|x| - j|y|$.

If there exists no such u, *then in the normal form* (2), $k = 1$ (x *is either primitive or weakly periodic*).

Proof. Suppose first that there exists u such that $x = u\{z\}u$. Observe that in general $|y| = |y'|$, and consider first the special case $|y| \geqslant |z|$. Because $u = yz = zy'$, we see that $x = yu = y^2z$. If $|y| > |z|$, it follows that the lemma holds with $j = 2$ and $y^* = z$; if $|y| = |z|$, then $y = z$ and the lemma holds with $j = 3$ and $y^* = \varepsilon$.

More generally, suppose that $|y| < |z|$. Since $u = yz = zy'$, it follows then that y is a prefix of z; in fact, y is a period of z and so repeats in u exactly $\lfloor |z|/|y| \rfloor + 1$ times. Since $x = yu$, the lemma holds in this case also.

Finally, observe that if $k > 1$, $u = (v^*v)^{k-1}v^*$ is a cover of x; thus, if no such cover exists, it follows that $k = 1$. □

We are now in a position to prove our main result:

Proof of Theorem 3. By Lemma 1, the theorem holds if x is primitive, since in this case x has no proper cover. If x is weakly periodic, then since as noted above every proper cover u of x satisfies $|u| \leqslant |v^*|$, we see that the theorem follows from Lemma 2. Thus for $k = 1$ the result holds. Observe also that every string u of the form (a), (b) or (c) is necessarily a cover of x. It therefore remains to be shown that, for $k > 1$, no strings u other than those specified in (a), (b) and (c) are covers of x.

Suppose therefore that $k > 1$ and that u is a cover of x such that

$$|(v^*v)^{j-1}v^*| < |u| < |(v^*v)^j v^*|, \tag{5}$$

where $j \in [1,k]$ if $v^* \neq \varepsilon$ and $j \in [2,k]$ otherwise. Observe then that two overlapping occurrences of u cover $(v^*v)^j v^*$, which therefore satisfies the conditions of Lemma 4. It follows that

$$(v^*v)^j v^* = y^{j'} y^*,$$

where y, y^* and z are defined as in Lemma 4 and $j' = \lfloor |z|/|y| \rfloor + 2$. But by (2.1), $|z| > |(v^*v)^{j-2}v^*|$ and $|y| < |v^*v|$, so that $j' \geqslant j$ and $|y^*| > |v^*|$. If $j' > j$, then we have found a second periodic breakdown of $(v^*v)^j v^*$, in contradiction to (1). If $j' = j$, then the string v^* chosen in (2) could not have been the string of maximum length, again a contradiction. (Here again, as with the impossibility of (3), the result is a special case of that given in [11].) We conclude that any choice of a cover u which satisfies (5) is impossible, so that only the strings specified by (a) are covers. □

As an example of this result, consider the string $x = ababaababababaaba$ [13]. This string has normal form $(ababaab)^2a$ with $v^* = a$, $v = babaab$ and $k = 2$. By Theorem 3(a), x has covers x and $(ababaab)a$; by Theorem 3(b), it also has as cover any cover of v^*vv^*, which has normal form $(ababa)aba$. By Theorem 3(c), this normal form gives rise to a third cover of x, namely aba. Since aba has no cover other than itself, the theorem also tells us that there is no other cover of x.

Theorem 3 implies that every string x contains a *minimal cover* which covers every other cover of x. This minimal cover is of course the cover of x of minimum length. A recent paper by Apostolico et al. [3] computes the minimal cover of x in linear time and space, while another by Breslauer [5] determines, also in linear space and time, whether or not a proper cover of x exists. The algorithm described in the next section computes solutions to both of these problems as byproducts of the computation of all the covers of x.

3. Computing all the covers of x

Theorem 3 essentially states that the covers of x are given *either* by case (a) of the theorem *or* by covers, which must also be of the form (a), of a proper prefix of x. Since in case (b) it is specified that $k > 1$, it follows that the prefix has length $|v^*vv^*| < 2n/3$; similarly, in case (c), we see that the length $|v^*| < n/2$. Thus Theorem

3 reduces a problem of size n to a problem of size less than $2n/3$, and this suggests the classical divide-and-conquer approach to the problem of finding all the covers of x. An initial formulation of such an approach is given by the procedure below. This procedure would be called by

COMPUTE_COVERS (x, k', PROPER);

(see Fig. 1) returning *either*

(1) all the covers of the string x (in the case that the integer argument $k' < 0$), *or*
(2) all those covers of x which are of length at most $k' \geqslant 0$.

The returned covers could include x itself only if the boolean argument PROPER were **false**.

Case (1) corresponds either to the initial call of COMPUTE_COVERS or to a recursive call for $k > 1$ as specified in Theorem 3(b). Case (2) corresponds to Theorem 3(c), where the covers found for v^* are required also to be covers of v^*vv^*: the procedure in effect supposes (and we show below) that this requirement can be reduced to the requirement that the covers of v^* are of length at most k'.

The output of this algorithm is encoded into a triple (v^*, v, newk), which gives all the information required to construct the *newk* covers of v determined at the current level of recursion. Note that the algorithm's final output necessarily includes an encoding of the minimal cover of x [3].

Consider first the execution time of procedure COMPUTE_COVERS. As we shall see below, the **output** operation and COMPUTE_NORMAL_FORM require only constant time. Thus the only other operation in COMPUTE_COVERS which could conceivably require more than constant time is the computation of the integer k' which we have represented by a function $k'(v^*vv^*)$. We shall see below that this function executes in time $\Theta(|v^*vv^*|)$, thus in time $O(n)$. Furthermore, since we shall see that the value returned by the function is zero or more, it follows that $k'(v^*vv^*)$ is executed at most once over all the recursive calls of COMPUTE_COVERS. We conclude that all but at most one of the recursive calls of COMPUTE_COVERS require only constant time; since there are at most $\log_{3/2} n$ such calls, it follows that COMPUTE_COVERS (x, k', PROPER) executes in time $O(n + \log n) = O(n)$. We devote the rest of this section to outlining an implementation that ensures this.

We deal first with the normal form of x. For integers i and j satisfying $1 \leqslant i \leqslant j \leqslant n$, let $x[i,j]$ denote the substring $x[i]x[i+1] \ldots x[j]$ of x. Then $x = x[1,n]$ and $x[i] = x[i,i]$. The *failure function* $f[i]$ of $x[1,i]$ is the length of the maximum border of $x[1,i]$; that is, the largest integer $j < i$ such that $x[1,j] = x[i-j+1,i]$. It is well known [1] that the failure functions $f[i]$, $i = 1, 2, \ldots, n$, corresponding to a given string x of length n can all be computed in total time $O(n)$. We suppose that this computation is performed by the procedure FAILURE_FUNCTION (x, f), executed exactly once before COMPUTE_COVERS is initially invoked.

Now compute $k_1 = n - f[n]$, and observe that in (2), $v^*v = x[1,k_1]$. Then the integer k defined in (1) is given by $k = n \operatorname{div} k_1$ and, setting $k_2 = n \bmod k_1$, we see that $v^* = x[1,k_2]$ and $v^*vv^* = x[1, k_1 + k_2]$. Thus the normal form (2) is completely determined by the triple (k_1, k_2, k) and is completely computable from n and $f[n]$. Moreover, the triple (k_1, k_2, k) makes available the same information as the triple (v^*, v, k), and thus may serve also as the output of COMPUTE_COVERS. We see then that COMPUTE_NORMAL_FORM and **output** each require only constant time, as claimed above.

Consider now Part I of COMPUTE_COVERS. Assuming still that $k' \geqslant 0$ can be correctly computed as the maximum length of a cover, it is easy to verify that the local variable *newk* is correctly computed as the maximum value of k such that

$$|v^*v|k + |v^*| = k_1k + k_2 \leqslant k'.$$

The **output** operation then identifies the covers $(v^*v)^j v^*$, $j = 1, 2, \ldots, \mathit{newk}$, in accordance with Theorem 3(a).

In Part II the first recursive call simply passes on the current value of k' and so correctly im-

```
procedure COMPUTE_COVERS ($x$: STRING; $k'$: integer; PROPER: boolean);
COMPUTE_NORMAL_FORM ($x, v^*, v, k$);

{Part I produces the output specified by Theorem 3(a).}
if $k' \geqslant 0$ then
  $newk \leftarrow (k' - |v^*|)$ div $|v^*v|$
else
  $newk \leftarrow k$;
if PROPER and ($newk = k$) then
  $newk \leftarrow newk - 1$;
if $newk > 0$ then
  output ($v^*, v, newk$);

{Part II executes the correct recursive call (or simply exits).}
if $k > 1$ then
  COMPUTE_COVERS ($v^*vv^*, k'$, true)        {Theorem 3(b).}
elsif $|v^*| \geqslant 3$ then
  {if $k' < 0$ then
     $k' \leftarrow k'(v^*vv^*)$;
   if $k' > 2$ then
     COMPUTE_COVERS ($v^*, k'$, false)}.      {Theorem 3(c).}
```

Fig. 1. Version 1.0.

plements Theorem 3(b) provided again that k' has been correctly computed. Based on the same assumption, the second recursive call is clearly also correct. It remains only to consider the exit conditions. Exit occurs either when the current string v^*, whose cover is to be computed in the next recursive call, has length 2 or less, or when the maximum length k' of any allowable cover is 2 or less. It is easy to verify that covers of length 2 or less arise only in trivial cases, and we have already seen that nontrivial covers of length 3 (for example, *aba*) can occur; thus the exit condition is also correct.

Before dealing with the computation of k', we re-express COMPUTE_COVERS in the light of these remarks, now using arguments and variables which may be given entirely in terms of the values of the failure functions $f[i]$, $i = 1, 2, \ldots, n$. The code to call COMPUTE_COVERS is given by

```
FAILURE_FUNCTION ($x, f$);
  {Compute the $f[i]$.}
COMPUTE_COVERS ($n, -1$, false);
  {Compute the covers of $x$.}
```

and the algorithm becomes as shown in Fig. 2.

We remark that it is trivial to remove, if desired, the recursion from this algorithm.

We turn now to the problem of computing k'. Recall that a cover u of v^* must be both a prefix and a suffix of v^*. Thus u is a cover of v^*vv^* if and only if, in the substring $vv^* = x[k_2 + 1, k_1 + k_2]$, there exist at most $|u| - 1$ consecutive positions i such that $f[i] < |u|$. Recall further that if u is a cover of v^*vv^*, then so is every cover of v^* of length less than $|u|$: we see that what we need to determine is the *greatest* integer k' such that in at most $k' - 1$ consecutive positions i of vv^*, $f[i] < k'$. In particular, if for any position i, $f[i] = 0 < 1$, then $k' = 0$ and no cover of v^* covers v^*vv^*.

The computation of k' takes place from left to right across the subarray $f[k_2 + 1, k_1 + k_2 - 1]$. The initial estimate is $k' = k_2$, a value to be successively reduced if at any stage k' consecutive positions of $f[k_2 + 1, k_1 + k_2 - 1]$ are found to contain no value $\geqslant k'$. The variable i' is used to store the position of the current rightmost occurrence of a value $\geqslant k'$. Since for every integer $i \in [2, n]$, $f[i] \leqslant f[i - 1] + 1$, it follows that if $f[i] < k'$, then no subsequent value k' can possibly be found before position $i + k' - f[i]$ of f. Thus k' is too large and should be reduced

```
procedure COMPUTE_COVERS ($n, k'$: integer; PROPER: boolean);
$k_1 \leftarrow n - f[n]$; $k_2 \leftarrow n \bmod k_1$; $k \leftarrow n \operatorname{div} k_1$;
if $k' \geqslant 0$ then
  $newk \leftarrow (k' - k_2) \operatorname{div} k_1$
else
  $newk \leftarrow k$;
if PROPER and $(newk = k)$ then
  $newk \leftarrow newk - 1$;
if $newk > 0$ then
  output $(k_1, k_2, newk)$;
if $k > 1$ then
  COMPUTE_COVERS ($k_1 + k_2, k'$, true)
elsif $k_2 \geqslant 3$ then
  {if $k' < 0$ then
     $k' \leftarrow k'(k_1, k_2)$;
   if $k' > 2$ then
     COMPUTE_COVERS ($k_2, k'$, false)}.
```

Fig. 2. Version 2.0.

if $i + k' - f[i] - 1 - i' \geqslant k'$; that is, if

$$i - i' > f[i]. \qquad (6)$$

Similar reasoning leads to the conclusion that the reduced value of k' should be set equal to the maximum value found in $f[i - f[i], i - 1]$. Hence we have:

```
function $k'(k_1, k_2$: integer): integer;
$k' \leftarrow k_2$; $i' \leftarrow k_2$; $i \leftarrow k_2$;
while $k' > 0$ and then $i < k_1 + k_2 - 1$ do
  {$i \leftarrow i + 1$;
  if $f[i] \geqslant k'$ then
    $i' \leftarrow i$
  elsif $i - i' > f[i]$ then    {See (6).}
    if $f[i] = 0$ then
      $k' \leftarrow 0$
    else
      {$k' \leftarrow$ maximum value in $f[i - f[i], i - 1]$;
       {See (7).}
       $i' \leftarrow$ rightmost position of $k'$}}.
```

There is a problem with this formulation of the function k': it is not clear that the recalculations of k' from $f[i - f[i], i - 1]$ can be carried out in time $O(n)$. There may a priori be as many as $k_2 - 2$ such recalculations for $k' = k_2 - 1, k_2 - 2, \ldots, 2$, and for each recalculation up to $k' - 1$ positions of f might need to be inspected: thus the total time requirement could be as much as $\Omega(k_2^2)$. Therefore, in order to be able to guarantee that the function k' executes in time $O(n)$, we introduce additional preprocessing: a procedure COMPUTE_MAXIMA (f, g, h) which computes integer arrays $g[1, n]$ and $h[1, n]$ from f. For every integer $i = 1, 2, \ldots, n$, we compute

$$g[i] = \max_{1 \leqslant j \leqslant f[i]} f[i - j]$$

and at the same time the corresponding $h[i]$, the rightmost position in $f[i - f[i], i - 1]$ at which $g[i]$ occurs. For this calculation, three distinct cases arise:

Case 1: $f[i] = 0$. In this case we set $g[i] \leftarrow 0$; $h[i] \leftarrow i$.

Case 2: $f[i] > f[i-1]$. Here $f[i] = f[i-1] + 1$, from which it follows that $f[i - f[i]] = f[(i-1) - f[i-1]]$. Hence

```
if $g[i-1] > f[i-1]$ then
  {$g[i] \leftarrow g[i-1]$; $h[i] \leftarrow h[i-1]$}
else
  {$g[i] \leftarrow f[i-1]$; $h[i] \leftarrow i - 1$};
```

Case 3: $f[i] \leqslant f[i-1]$. In view of (1), we may suppose that $f[i] > 0$. Hence

$$x[i - f[i-1], i] \\ = (u_1 u_2 \ldots u_p)^j (u_1 u_2 \ldots u_{p'}),$$

where

$$p = f[i-1] + 1 - f[i],$$
$$j = (f[i-1] + 1) \operatorname{div} p,$$
$$p' = (f[i-1] + 1) \bmod p.$$

That is, $x[i - f[i-1], i]$ is a periodic substring with period of length p whose initial $f[i-1]$ characters are a prefix of x. It follows that $f[i-1]$ is a maximum in $f[i - f[i-1], i-1]$, and so we set $g[i] \leftarrow f[i-1]$, $h[i] \leftarrow i-1$.

The processing required to compute $g[i]$ and $h[i]$ for each $i = 1,2,\ldots,n$ is constant in each of these three cases, and so COMPUTE_MAXIMA is an $O(n)$ algorithm. (Indeed, the computation of g and h may if desired conveniently be incorporated into the procedure FAILURE_FUNCTION.) Thus, with the sequence of instructions

```
FAILURE_FUNCTION$(x, f)$;
COMPUTE_MAXIMA$(f, g, h)$;
COMPUTE_COVERS$(n, -1,$ **false**$)$;
```

we may call Version 2.0 of COMPUTE_COVERS and make use of the function $k'(k_1, k_2)$ to compute all the covers of x in time $O(n)$. To do this, we need only replace the last two lines of function k' with

$$\{k' \leftarrow g[i];\ i' \leftarrow h[i]\}. \tag{7}$$

Observe that, with this replacement, the function k' executes in time $O(k_1)$; as discussed above, every other calculation carried out in Version 2.0 requires only constant time. We then state formally the main result of this paper:

Theorem 5. *All the covers of a given string x of length n may be computed using time $\Theta(n)$ and additional space $\Theta(n)$.* □

4. Concluding remarks

In this paper we have characterized the covers of a given string x of length n in terms of a normal form for x (Theorem 3). This characterization has enabled us to compute all the covers of x in linear time (Theorem 5). In [9] Iliopoulos et al. define a substring u of x to be a *seed* of x iff u is a cover of a string w which contains x as a substring. Thus every cover of x is also a seed of x; and, for example, *abc*, *bca* and *cab* are all seeds of $x = abcab$. In [9] an $O(n \log n)$ algorithm to compute all the seeds of x is presented: it appears possible that Theorem 3 and related results can be used to simplify or reduce the complexity of this algorithm.

The results of this paper may also be applicable to a problem studied by Apostolico and Ehrenfeucht [2]: the computation of all substrings of x which have a proper cover. (This is a generalization of the problem of computing all the repetitions in x [6].) In [2] suffix trees are used to compute all of these substrings in time $O(n \log^2 n)$.

Acknowledgement

The work of the second author was supported in part by Grant No.A8180 of the Natural Sciences and Engineering Research Council of Canada. The authors are pleased to acknowledge the contribution of Kunsoo Park, who found an error in the original formulation of Theorem 3 and also contributed the example quoted at the end of Section 2.

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 and A. Ehrenfeucht, Efficient detection of quasiperiodicities in strings, Tech. Rept. No. 90.5, The Leonardo Fibonacci Institute, Trento, Italy, 1990.

[3] A. Apostolico, M. Farach and C.S. Iliopoulos, Optimal superprimitivity testing for strings, *Inform. Process. Lett.* **39** (1) (1991) 17–20.

[4] R.S. Boyer and J.S. Moore, A fast string searching algorithm, *Comm. ACM* **20** (10) (1977) 62–72.

[5] D. Breslauer, An on-line string superprimitivity test, *Inform. Process. Lett.* **44** (6) (1992) 345–347.

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

[7] L.J. Cummings, Overlapping substrings and Thue's problem, in: *Proc. 3rd Caribbean Conf. on Combinatorics and Computing*, University of the West Indies, Cave Hill (1981) 99–109.
[8] L.J. Cummings, Strongly qth power-free strings, *Ann. Discrete Math.* **17** (1983) 247–252.
[9] C.S. Iliopoulos, D. Moore and K. Park, Covering a string, in: *Proc. 4th Ann. Symp. on Combinatorial Pattern Matching* (1993) 54–62.
[10] D.E. Knuth, J.H. Morris and V.R. Pratt, Fast pattern matching in strings, *SIAM J. Comput.* **6** (2) (1977) 323–350.
[11] R.C. Lyndon and M.P. Schutzenberger, The equation $a^M = c^N c^P$ in a free group, *Michigan Math. J.* **9** (1962) 289–298.
[12] D. Moore and W.F. Smyth, Computing the covers of a string in linear time, in: *Proc. 5th Ann. ACM–SIAM Symp. on Discrete Algorithms* (1994) 511–515.
[13] K. Park, Private communication.