Chapter 1
Computing the Covers of a String in Linear Time

Dennis Moore* W. F. Smyth†

1 Introduction.

Let x denote a given nonempty string of length $n = |x| \geq 1$. A string u is a *cover* of x iff every position of x occurs within an occurrence of u within x. Thus x is always a cover of itself, and, for example, $u = aba$ is a cover of $x = ababaaba$. A cover $u \neq x$ is called a *proper cover*. Covers and their generalizations are an interesting extension of the idea of a "repetition" [4]; they have potential applications in DNA sequence analysis, and have recently been studied in [1], [2], [7]. 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 $\Theta(n)$. By avoiding unnecessary checks, this algorithm appears to execute more quickly than that given in [2].

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

$$x = (v'v)^k v' \tag{1.1}$$

over all possible choices of strings v' (possibly empty) and $v \neq \epsilon$. Then let v^* be the maximum length string v' which satisfies (1.1). Thus

$$x = (v^*v)^k v^*. \tag{1.2}$$

If $v^* = \epsilon$, then we say that x is either *primitive* (for $k = 1$) or a *repetition* (for $k > 1$); if $v^* \neq \epsilon$, we say that x is either *weakly periodic* (for $k = 1$) or *strongly periodic* (for $k > 1$). We call the righthand side of (1.2) the *normal form* of x; in §3 we show how the normal form can be efficiently computed.

For all strings y and z such that $x = yz$, we say that y is a *prefix* and z a *suffix* of x; if y and z are both nonempty, then we say that both suffix and prefix are *proper*. A string z which is both a proper prefix and a proper suffix of x is called a *border* of x. Thus in (1.1), if $v' \neq \epsilon$, then x has border v'; we see in fact that a nonempty string is primitive if and only if it has no border. Further, we see that every string with a proper cover u necessarily has border u, an observation which yields

LEMMA 1.1.1. *No primitive string has a proper cover.*

Also, it is not difficult to show that

LEMMA 1.1.2. *If u is a proper cover of a string x that is not a repetition, then u is not primitive.*

In the next section we state two less obvious properties of covers, results that can then be used to establish a characterization theorem for covers. In §3, based on this characterization, we present a simple linear time algorithm which computes all the covers of a given string x. §4 briefly discusses future work.

2 A Characterization of Covers.

We state two preliminary results, the first a special case of a result due to Cummings [5], [6]:

LEMMA 1.2.1. *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 (1.2), $k = 1$ (x is either primitive or weakly periodic).

LEMMA 1.2.2. *Let u be a proper cover of x, and let $z \neq u$ be a substring of x such that $|z| \leq |u|$. Then z is a cover of x if and only if z is a cover of u.*

When a longest proper cover u of x has already been found, this second lemma reduces the problem of finding shorter covers of x to the problem of finding covers of u. In particular, for $k = 1$, it follows that every proper cover of x is a cover of v^*. More generally, using the lemmas stated above, we find that we can characterize the covers of x as follows:

THEOREM 1.2.1. *Each cover u of x is* **either**
(a) $(v^*v)^j v^*$, $j = 1, 2, \ldots, k$; **or**
(b) (if $v^* = \epsilon$) a proper cover of v; **or**
(c) (if $v^* \neq \epsilon$) a cover of v^* which also covers v^*vv^*.

Proof. By Lemma 1.1.1, the theorem holds if x is primitive. If x is weakly periodic, then since every cover u of x satisfies $|u| \leq |v^*|$, we see that the result follows from Lemma 1.2.2. Observe also that every string u of the

* Curtin University of Technology, Perth, Western Australia.

† McMaster University, Hamilton, Ontario and Curtin University; supported by Grant No. A8180 of the Natural Sciences & Engineering Research Council of Canada.

form (a), (b) or (c) is necessarily a cover of x. Thus it 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. If v^* is empty, it is an easy consequence of Lemma 1.2.2 that every cover u satisfying $|u| < |v|$ must also satisfy (b). If v^* is not empty and $|u| \leq |v^*|$, then Lemma 1.2.2 may again be used to show that u satisfies (c). Hence suppose that

$$|(v^*v)^{j-1}v^*| < |u| < |(v^*v)^j v^*|, \tag{1.3}$$

where $j \in [1,k]$ if $v^* \neq \epsilon$ and $j \in [2,k]$ otherwise. The argument to be presented here deals with the case of nonempty v^*; the proof for $v^* = \epsilon$ is similar. Observe then that two overlapping occurrences of u cover $(v^*v)^j v^*$, which consequently satisfies the conditions of Lemma 1.2.1. It follows that

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

where y, y^* and z are defined as in Lemma 1.2.1 and $j' = \lfloor |z|/|y| \rfloor + 2$. But by (1.3), $|z| > |(v^*v)^{j-2}v^*|$ and $|y| < |v^*v|$, so that $j' \geq j$. If $j' > j$, then we have found a second periodic breakdown of $(v^*v)^j v^*$, in contradiction to (1.1). If $j' = j$, then the string v^* chosen in (1.2) could not have been the string of maximum length, again a contradiction. (The argument here is a special case of the well-known result [8] that a string can have at most one primitive period.) We conclude that any choice of a cover u which satisfies (1.3) is impossible, so that only the strings specified by (a) are covers.

As an example of this result, consider the string $x = abaababaababa$. The normal form of x is $(abaab)^2 aba$ with $v^* = aba$, $v = ab$ and $k = 2$. By Theorem 1.2.1(a), x has covers x and $(abaab)aba$; by Theorem 1.2.1(c), it has the cover $v^* = aba$ which covers v^*vv^*; since v^* has no cover other than itself, the theorem tells us that there is no other cover of x.

3 Computing All Covers of x.

A consequence of Theorem 1.2.1 is 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) a proper cover of v can exist only if $k > 1$, it follows that the prefix has length $|v| \leq n/2$; similarly, in case (c), we see that the length $|v^*| < n/2$. Thus Theorem 1.2.1 reduces a problem of size n to a problem of size at most $n/2$, 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

$$\text{COMPUTE\_COVERS } (x, \epsilon);$$

where x is the string whose covers are required; corresponding to case (c) of Theorem 1.2.1, such a call might give rise to a recursive call

$$\text{COMPUTE\_COVERS } (v^*, v^*vv^*);$$

where now, since the second parameter is nonempty, the covers found for v^* are required also to be covers of v^*vv^*.

```
procedure COMPUTE_COVERS (x, w : STRING);
COMPUTE_NORMAL_FORM (x, v*, v, k);
if w = ε then
  output (v*, v, k)
else
  output strings (v*, v, j), 1 ≤ j ≤ k, which cover w;
if not FINISHED then
  if |v*| = 0 then
    COMPUTE_COVERS (v, ε)
  else
    COMPUTE_COVERS (v*, v*vv*).
```

Version 1.0

The output of this algorithm is encoded into a triple (v^*, v, k), which gives all the information required to construct the k covers of v determined at the current level of recursion. We have referred above to the requirement to check that covers of v^* cover v^*vv^*. Observe that, if this check can be performed in time $O(|x|)$, and if in addition COMPUTE_NORMAL_FORM requires only $O(|x|)$ time, then Version 1.0 will execute in time at most proportional to $|x| + |x|/2 + |x|/4 + \cdots < 2n$; thus COMPUTE_COVERS would be an $O(n)$ algorithm. We now go on to show how this linear time bound can be achieved.

We deal first with the normal form of x. For integers i and j satisfying $1 \leq i \leq j \leq n$, 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 [3] that the failure functions $f[i]$, $i = 1, 2, \ldots, n$, corresponding to a given string

x of length n can be computed in time $O(n)$. We suppose that this computation is performed by the procedure FAILURE_FUNCTION (x, f).

Now compute $k_1 = n - f[n]$, and observe that in (1.2), $v^*v = x[1, k_1]$. Then the integer k defined in (1.1) is given by $k = n$ 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 (1.2) 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 serves also as the output of COMPUTE_COVERS.

Suppose now that COMPUTE_COVERS is called with nonempty w (and x a substring of w). Lemma 1.2.2 tells us that if a cover of x covers w, then so does every shorter cover of x. More specifically, it follows from Theorem 1.2.1 that there exists an integer k' such that $u = (v^*v)^j v^*$, $j = 1, 2, \ldots, k$, covers w if and only if $|u| \leq k'$. We discuss later the computation of k'; for the moment, we suppose that $k' = k'(w)$ is computable as a function of w.

To implement the Boolean expression FINISHED, observe that if x is primitive ($k = 1$ and $k_2 = 0$), all covers have already been computed, and so exit from the current level of recursion may take place. The recursion should also end in the case that the current recursive call is for a prefix v^* (so that w is nonempty) and no output has occurred (no cover of v^* covered w).

Putting together these remarks, assuming that f is available as a global variable, and making minor adjustments to eliminate the possibility of duplicate outputs (by returning, in the normal case, proper covers only), we find that a new version of COMPUTE_COVERS 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 then given by

```
FAILURE_FUNCTION (x, f);   {Compute the f[i].}
output (n, 0, 1);          {Output x itself.}
COMPUTE_COVERS (n, -1);    {Proper covers only.}
```

and the algorithm becomes

```
procedure COMPUTE_COVERS (n, k' : integer);
var
  k1, k2, k : integer;
k1 <-- n - f[n]; k2 <-- n mod k1; k <-- n div k1;
if k' < 0 then
  k <-- k - 1
else
  recompute k based on k';          {See below, (1.4).}
  k <-- min(k, max(0, k' - k2) div k1);
if k ≠ 0 then
  {output (k1, k2, k);
  if k2 = 0 then
    COMPUTE_COVERS (k1, -1)
  else
    COMPUTE_COVERS (k2, k'(k1, k2))}
                                     {See below.}
elsif k2 > 0 and k' = -1 then
  COMPUTE_COVERS (k2, k'(k1, k2)).
```

Version 2.0

In order to give a still more precise specification of the implementation of our algorithm, we consider first the recomputation of k from k'. Suppose that procedure COMPUTE_COVERS has been called with actual parameters $n = \kappa_2$ and $k'(\kappa_1, \kappa_2)$ corresponding to $x = x[1, \kappa_2]$ and $w = x[1, \kappa_1 + \kappa_2]$. Then in the current level of recursion, we compute k_1, k_2 and k corresponding to the normal form

$$x[1, \kappa_2] = x[1, k_1]^k x[1, k_2].$$

In order that $x[1, k_1]^j x[1, k_2]$, $1 \leq j \leq k$, should be a cover of w, it must be true that $jk_1 + k_2 \leq k'$; that is, that

$$j \leq (k' - k_2)/k_1.$$

This relationship permits us to recompute k, as stated in Version 2.0, by substituting the following single line of code:

$$k \longleftarrow \min(k, \max(0, k' - k_2) \text{ div } k_1); \tag{1.4}$$

We turn now to the more difficult problem of computing k' itself.

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 $\geq k'$. The variable i' is used to store the position of the current rightmost occurrence of a value $\geq k'$. Since for every integer $i \in [2,n]$, $f[i] \leq f[i-1]+1$, 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 if $i+k'-f[i]-1-i' \geq k'$; that is, if

$$i - i' > f[i]. \tag{1.5}$$

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

function $k'(k_1, k_2$: **integer**) : **integer**;
var
  i, i' : **integer**;
$k' \longleftarrow k_2$; $i' \longleftarrow k_2$; $i \longleftarrow k_2$;
while $k' > 0$ **and then** $i < k_1 + k_2 - 1$ **do**
  $\{i \longleftarrow i+1$;
  if $f[i] \geq k'$ **then**
    $i' \longleftarrow i$
  elsif $i - i' > f[i]$ **then**   {See above, (1.5).}
    if $f[i] = 0$ **then**
      $k' \longleftarrow 0$
    else
      $\{k' \longleftarrow$ maximum value in $f\big[i-f[i], i-1\big]$;
          {See below, (1.6).}
      $i' \longleftarrow$ 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 $f\big[i-f[i], i-1\big]$ 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

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

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

(1) $f[i] = 0$.
In this case we set $g[i] \longleftarrow 0$; $h[i] \longleftarrow i$.

(2) $f[i] > f[i-1]$.
Here $f[i] = f[i-1]+1$, from which it follows that $f\big[i-f[i]\big] = f\big[(i-1)-f[i-1]\big]$. Hence
if $g[i-1] > f[i-1]$ **then**
  $\{g[i] \longleftarrow g[i-1]$; $h[i] \longleftarrow h[i-1]\}$
else
  $\{g[i] \longleftarrow f[i-1]$; $h[i] \longleftarrow i-1\}$;

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

$$x\big[i-f[i-1], i\big] = (u_1u_2\ldots u_p)^j(u_1u_2\ldots u_{p'}),$$

where

$$p = f[i-1]+1-f[i];$$
$$j = \big(f[i-1]+1\big) \text{ div } p;$$
$$p' = \big(f[i-1]+1\big) \text{ mod } p.$$

That is, $x\big[i-f[i-1], i\big]$ 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 $f\big[i-f[i-1], i-1\big]$, and so we set $g[i] \longleftarrow f[i-1]$; $h[i] \longleftarrow i-1$.

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);
output $(n, 0, 1)$;
COMPUTE_COVERS $(n, -1)$;

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

$$\{k' \longleftarrow g[i];\ i' \longleftarrow h[i]\}. \tag{1.6}$$

Observe that, with this replacement, the function k' executes in time $O(k_1+k_2)$; every other calculation carried out in Version 2.0 of COMPUTE_COVERS requires only constant time. We then state formally the main result of this paper:

THEOREM 1.3.1. *Version 2.0 computes all the covers of a given string x of length n in time $\Theta(n)$ and space $\Theta(n)$.*

4 Concluding Remarks.

In [7] Iliopoulos *et al.* adopt a more general definition of "cover", which we call here a "generator": we say that a substring u of x is a *generator* 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 generator of x; and, for example, abc, bca and cab are all generators of $x = abcab$. [7] presents an $O(n \log n)$ algorithm to compute all the generators of x: it appears possible that Theorem 1.2.1 and related results can be used to simplify or reduce the complexity of their algorithm.

The results of this paper may also be applicable to a problem studied by Apostolico and Ehrenfeucht [1]: 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 [4].) [1] uses suffix trees to compute all of these substrings in time $O(n \log^2 n)$.

References

[1] Alberto Apostolico & Andrzej Ehrenfeucht, *Efficient detection of quasiperiodicities in strings*, submitted for publication.

[2] Alberto Apostolico, Martin Farach & Costas S. Iliopoulos, *Optimal superprimitivity testing for strings*, submitted for publication.

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

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

[5] L. J. Cummings, *Overlapping substrings and Thue's problem*, Proc. Third Caribbean Conf. Combinatorics & Computing, University of the West Indies, Cave Hill (1981) 99-109.

[6] L. J. Cummings, *Strongly qth power-free strings*, Annals of Discrete Mathematics 17 (1983) 247-252.

[7] Costas S. Iliopoulos, Dennis Moore & Kunsoo Park, *Covering a string*, submitted for publication.

[8] R. C. Lyndon & M. P. Schutzenberger, *The equation $a^M = c^N c^P$ in a free group*, Michigan Mathematical Journal 9 (1962) 289-298.