COMPUTING THE COVER ARRAY IN LINEAR TIME

Yin Li
Department of Computing & Software
McMaster University
&
IBM Canada

W. F. Smyth[*]
Department of Computing & Software
McMaster University
&
School of Computing
Curtin University of Technology
e-mail: `smyth@mcmaster.ca`

July 6, 1999

ABSTRACT

Let x denote a given nonempty string of length $n = |x|$. A proper substring u of x is a *proper cover* of x if and only if every position of x lies within an occurrence of u within x. This paper introduces an array $\gamma = \gamma[1..n]$ called the *cover array* in which each element $\gamma[i]$, $1 \le i \le n$, is the length of the longest proper cover of $x[1..i]$ or zero if no such cover exists. In fact it turns

[*] Communicating author.

out that γ describes *all* the covers of every prefix of x. Several interesting properties of γ are established, and a simple algorithm is presented that computes γ on-line in $\Theta(n)$ time using $\Theta(n)$ additional space. Thus the new algorithm computes for all prefixes of x information that previous cover algorithms could compute only for x itself, and does so with no increase in space or time complexity.

KEYWORDS

string, word, algorithm, period, cover

1 INTRODUCTION

Let x denote a nonempty *string* of *length* $n \geq 1$. A substring u of x is called a *cover* of x (also, u *covers* x) if and only if x can be constructed by concatenating or overlapping copies of u, so that every position of x lies within an occurrence of u within x. Thus x is always a cover of itself; if a proper substring u of x is also a cover of x, then u is called a *proper cover* of x. For example, the string $x = abcabcaabca$ has covers x and $abca$, and proper cover $abca$. A string that has a proper cover is called *coverable.*

The idea of a coverable string generalizes the idea of a *repetition*; that is, a string x that can be constructed by concatenating copies of some proper substring u of x. Repetitions in strings were first studied by the mathematician Axel Thue [17], who showed how to construct repetition-free strings of unbounded length on three letters. Over the last two decades, computer scientists have become interested in the algorithmic problem of computing all the repetitions in a given string [8, 14, 4], a task that requires $\Theta(n \log n)$ time. It is interesting to note that Thue also showed in [17] how to construct infinite strings without overlaps — that is, with no coverable substrings that are not repetitions. The corresponding algorithmic problem, the computation of the maximal coverable substrings of a given string, has also been solved in modern times [1, 10].

Repetitions and covers in strings are special cases of "approximate periodicity", a topic that has potential applications in molecular biology, probability theory, coding theory, data compression, and formal language theory. In fact, a generalization of the idea of a cover provides a basis for classifying strings based on a kind of approximate periodicity: given x and a positive integer $k \leq n$, a set $U_k = \{u_1, u_2, \ldots, u_m\}$ of strings of length k is said to be a *k-cover* of x if and only if x can be constructed by concatenating or overlapping elements of U_k. The k-cover is said to be *minimum* if and only if $m = |U_k|$ is a minimum for given x and k. Thus for a range of values of k, the minimum k-cover can provide a measure of how close to periodic every string x is. An algorithm that computes a minimum k-cover in $O\big(n^2(n-k)\big)$ time has recently been proposed [12].

While the problem of determining whether or not a given string x is a repetition is trivial, the problem of determining whether or not x has a cover is not. In [2] Apostolico, Farach and Iliopoulos presented a linear-time algorithm for computing the shortest cover of x. Then in [6] Breslauer published an on-line algorithm for the same problem. More recently, Moore and Smyth [15, 16] showed how to compute all the covers of x in linear time, a result considerably extended in this paper with an algorithm that computes on-line in linear time all the covers of every prefix of x. PRAM algorithms have also been developed for the shortest cover [7] and all-covers [11] problems, both dependent on an efficient algorithm for the subtree max gap problem [5].

Like the previous algorithms for computing covers of x, our algorithm makes use of the *border array* of x; that is, the array $\beta = \beta[1..n]$ that gives in each position i the length of the longest border of $x[1..i]$. In addition, we introduce, analogous to the border array, a "cover array" and corresponding "cover tree". The new algorithm also is closely analogous to the well-known border array algorithm [3]; for each i, it has two main components:

- update the current position i in the cover array — that is, insert i in its correct position in the cover tree;
- compute positions $j < i$ in the cover array such that $x[1..j]$ cannot be a cover of any $x[1..k]$, $k \geq i$.

In Section 2 we introduce the cover array and establish some of its properties. In Section 3 we describe our new algorithm.

2 THE COVER ARRAY

We let x denote a string of length $|x| = n$ and represent it as an array $x[1..n]$. The special symbol ϵ denotes the *empty string*; that is, the string of length

0. A proper substring u of x that is both a prefix and a suffix of x is called a *border* of x. Thus every string has the empty border and, for example, $x = abaabaab$ has the borders $u = abaab$, $u = ab$ and $u = \epsilon$. As observed in the introduction, a border array β is used to store the length of the longest border of each $x[1..i]$; for example, the border array of *abaabaab* is 0011234. For a more elaborate example, see Figure 1.

It is convenient to introduce the notation $\beta^1[i] = \beta[i]$, $1 \le i \le n$, with $\beta^j[i] = \beta\big[\beta^{j-1}[i]\big]$ for every $j \ge 2$ such that $\beta^{j-1}[i]$ is defined and $1 \le \beta^{j-1}[i] \le n$. Then $x\big[1..\beta[i]\big]$ is the longest border of $x[1..i]$, and $x\big[1..\beta^j[i]\big]$ is the longest border of $x\big[1..\beta^{j-1}[i]\big]$ for every $j \in 2..m$. Since by definition $0 \le \beta[i] < i$ for every $i \in 1..n$, it follows that the sequence

$$\beta[i], \beta^2[i], \ldots, \beta^m[i] \qquad (1)$$

is well-defined for every i and monotone decreasing to $\beta^m[i] = 0$ for some $m \ge 1$. It is well known [3] that this sequence in fact identifies *every* border of $x[1..i]$, and further that, given x, β can be computed on-line in $\Theta(n)$ time (using the so-called "failure function" algorithm). We quote without proof another well-known result [3] that will be required later:

Lemma 2.1 *For every border array $\beta[1..n]$ and every integer $i \in 1..n-1$, the only possible values of $\beta[i+1]$ are zero and $b+1$, where b denotes any element of the sequence (1).* □

Corresponding to each $\beta^j[i]$, $1 \le j \le m$, we define a *period* $i - \beta^j[i]$ of $x[1..i]$. The longest period corresponding to $j = 1$ is called *the period* of x. If $p = i - \beta[i]$, $r = \lfloor i/p \rfloor$, $q = i \bmod p$, $u = x[1..p]$ and $u' = x[1..q]$, then we say that $x[1..i] = u^r u'$ is written in *normal form.*

It is useful to represent the border array β as a tree T_β, called the *border tree* [9, 16]; that is, a rooted tree in which each node has a unique integer label chosen from $0..n$ and the following rules hold: the root has label 0, and the parent of the node with label i, $i = 1, 2, \ldots, n$, is the node with label $\beta[i]$. Then as illustrated in Figure 1, the labels of the ancestors of the node labelled i are exactly the lengths of the borders of $x[1..i]$.

The concept of the cover array is very similar to that of the border array and, as we shall see below, especially in Theorems 2.2 and 2.3, the values of corresponding elements in these two arrays are closely related too. We let $\gamma = \gamma[1..n]$ denote the *cover array* of string x, where each element $\gamma[i]$ specifies the length of the longest proper cover of $x[1..i]$ or zero if there is no

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
$x =$	a	b	a	a	b	a	b	a	a	b	a	a	b	a	b	a	a	b	a	b	a	b	a
$\beta =$	0	0	1	1	2	3	2	3	4	5	6	4	5	6	7	8	9	10	11	7	8	2	3
$\gamma =$	0	0	0	0	0	3	0	3	0	5	6	0	5	6	0	8	9	10	11	0	8	0	3

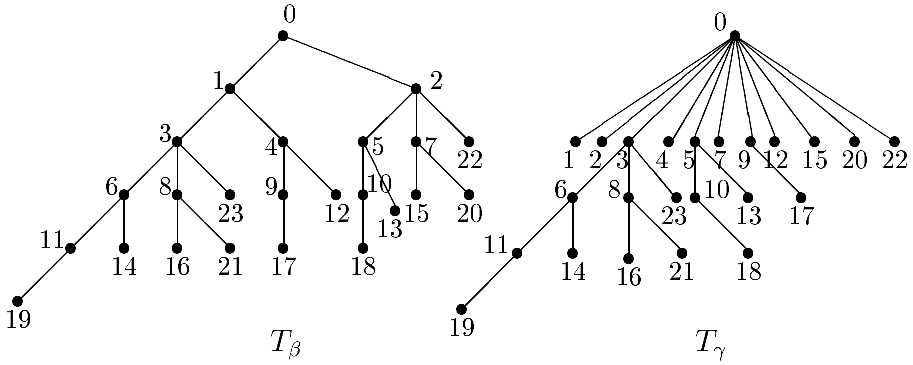

Figure 1: **The Border and Cover Trees of x.**

proper cover. It is shown in [6] that if u_1 and u_2 are covers of x, $|u_1| > |u_2|$, then u_2 is a cover of u_1. Thus, analogous to the sequence (1) that exists for each position i in the border array, we find in the cover array a corresponding monotone decreasing sequence

$$\gamma[i], \gamma^2[i], \ldots, \gamma^m[i] \tag{2}$$

with $\gamma^m[i] = 0$, defined for every $i \in 1..n$. This sequence identifies in descending order of length all the proper covers

$$x\big[1..\gamma[i]\big], x\big[1..\gamma^2[i]\big], \ldots, x\big[1..\gamma^{m-1}[i]\big]$$

of every $x[1..i]$.

Analogous to T_β, a *cover tree* T_γ is a rooted tree representing γ in which each node has a unique integer label chosen from $0..n$, the root has label 0, and the parent of the node with label i, $i = 1, 2, ..., n$, is the node with label $\gamma[i]$. Then the nonzero labels of the ancestors of the node labelled i are exactly the lengths of the proper covers of $x[1..i]$, which therefore can be determined simply by visiting i's ancestors in T_γ. See Figure 1.

We now investigate properties of the cover array and its relation to the border array. Of course, since a proper cover must be a border, we see that every $\gamma[i]$ must assume one of the values (1). Hence $\gamma[i] \le \beta[i]$ for all i

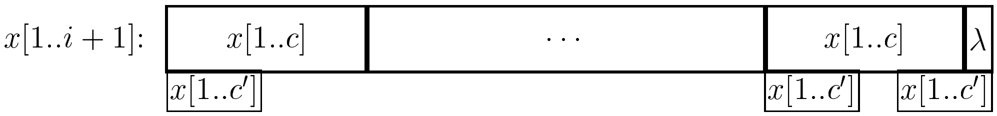

Figure 2: $x[1..c']$ **is a proper cover of** $x[1..c]\lambda$**.**

and, in particular, $\beta[i] = 0$ implies $\gamma[i] = 0$. But it turns out that there are much tighter, and much less obvious, restrictions on the values that may be assumed by $\gamma[i]$, as the next two theorems show.

Theorem 2.2 *For every integer* $i \in 1..n-1$, *if* $\gamma[i] \neq 0$, *then either* $\gamma[i+1] = \gamma[i] + 1$ *or* $\gamma[i+1] = 0$.

Proof Let $c = \gamma[i]$. Then $c > 0$ and $x[1..c]$ is the longest cover of $x[1..i]$. Suppose the theorem is false, so that there exists a positive integer $c' \leq c$ such that $x[1..c']$ is the longest cover of $x[1..i+1] = x[1..i]\lambda$. Then (see Figure 2) it follows that $x[1..c']$ is a proper cover of $x[1..c]\lambda$. Observe however that, for every $\lambda' \neq \lambda$, $x[1..c']$ is not a cover of $x[1..c]\lambda'$.

If every occurrence of $x[1..c]$ in $x[1..i+1]$ were followed by λ, it would follow that $x[1..c+1]$ would be a cover of $x[1..i+1]$, contrary to the hypothesis that $x[1..c']$ is the longest cover. Thus some occurrence of $x[1..c]$ in $x[1..i+1]$ must be followed by $\lambda' \neq \lambda$. But as we see below, in Lemma 2.4, this is impossible: it turns out that *every* occurrence of $x[1..c]$ in $x[1..i]$ must be followed by λ. We conclude that $x[1..c+1]$ must be a cover of $x[1..i+1]$, and so the assumption $0 < c' \leq c$ is false, and the theorem is proved. □

Theorem 2.3 *For every integer* $i \in 1..n-1$, *if* $\beta[i+1] \leq \beta[i]$, *then* $\gamma[i+1] = 0$.

Proof Since $\beta[i+1] = 0$ implies $\gamma[i+1] = 0$, we may assume without loss of generality that $\beta[i+1] > 0$. Then by Lemma 2.1, $\beta[i+1] = \beta^j[i] + 1$ for some positive integer j; by hypothesis, $j \geq 2$, so that $\beta[i] > 0$. Suppose that the theorem is false, so that $\gamma[i+1] = c' > 0$ and $x[1..c']$ is a cover of

$x[1..i+1]$. Then, setting $c = \beta[i]$, we have

$$0 \leq c' = \gamma[i+1] \leq \beta[i+1] \leq c.$$

Now let $x[i+1] = \lambda$. Since $\beta[i+1] \neq \beta[i]+1$, it must be true that $x[c+1] \neq \lambda$; that is, an occurrence of $x[1..c]$ in $x[1..i+1]$ is followed by some $\lambda' \neq \lambda$, again contrary to Lemma 2.4. Therefore the assumption that $\gamma[i+1] > 0$ must be false. □

Both of the preceding theorems depend on the following lemma. Unfortunately, the only proof of it that the authors have been able to devise is rather technical.

Lemma 2.4 *Suppose that a nonempty string $x = x[1..i]$ has a nonempty border $x[1..c]$. Suppose further that $x[i+1] = x[i]\lambda$ has a proper cover $x[1..c']$ for some $c' \leq c$. Then for every letter $\lambda' \neq \lambda$, the substring $x[1..c]\lambda'$ does not occur in $x[1..i]$.*

Proof Suppose on the contrary that a substring $x[1..c]\lambda'$ does exist in $x[1..i]$ for some $\lambda' \neq \lambda$. Then as shown in Figure 3, since $x[1..c']$ is a cover, two occurrences of $x[1..c']$ must overlap, and $x[1..c'-1] = x[c-c'+2..c]$. Hence there exist positive integers h and h' such that

$$\begin{gathered} x[1..h] = x[c'-h+1..c'] = x[c'-h+1..c'-1]\lambda, \\ x[1..h'] = x[c'-h'+1..c'-1]\lambda', \end{gathered}$$

where $h + h' \geq c'$. It is implied that $h \neq h'$, since otherwise we have $\lambda = \lambda'$, contrary to the hypothesis. Observe that $x[1..h'-1]$ and $x[1..h-1]$ are both borders of $x[1..c'-1]$. Then

$$(h-1) + (h'-1) \geq (c'-1) - 1. \tag{3}$$

Without loss of generality, suppose that $h \geq h'+1$, so that (3) becomes

$$h - 1 \geq (c'-1)/2$$

and $x[1..c'-1]$ has a border of length at least $(c'-1)/2$. Hence we may write $x[1..c'-1]$ in the normal form

$$u^k = u^{\lfloor k \rfloor}u^*, \ k \geq 2,$$

where u^* is a possibly empty proper prefix of u, and u is not a repetition.

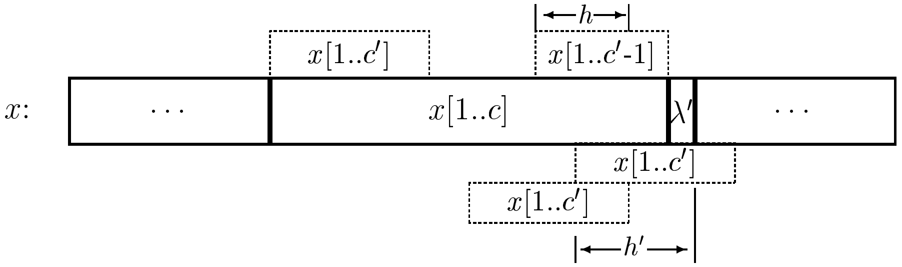

Figure 3: **Two occurrences of $x[1..c']$ must overlap.**

Consider now an occurrence of $x[1..c'-1]\lambda' = u^{\lfloor k\rfloor}u^*\lambda'$, with overlapping strings $x[1..c'-1]\lambda = u^{\lfloor k\rfloor}u^*\lambda$ at left and right, as shown in Figure 3. Since u is not a repetition, it is well known [13] that u therefore does not equal any nontrivial rotation of itself; that is, if we write $u = u_{PRE}u_{SUF}$ for nonempty u_{PRE} and u_{SUF}, then $u \neq u_{SUF}u_{PRE}$. Further, since $u^*\lambda$ does not match with $u^*\lambda'$, the only possible overlap at the left is of the form

$$\begin{array}{r|l|l} \overbrace{u\,u\ldots}^{k-1} & u & u^*\lambda' \\ \underbrace{uu\ldots u}_{k} & u^*\lambda & , \end{array}$$

where $u^*\lambda$ overlaps the first $|u^*\lambda|$ positions of an occurrence of u in $x[1..c'-1]$. It follows that $u^*\lambda$ is a prefix of u, while $u^*\lambda'$ is not.

Hence the overlap at the right *cannot* take the form:

$$\begin{array}{r|l|l} \overbrace{uu\ldots u}^{k} & u^*\lambda' & \\ & u & \underbrace{u\ldots u}_{k-1}u^*\lambda, \end{array}$$

and so must occur as follows:

$$\begin{array}{r|l|l} uu\ldots & \hat{u} & u^*\lambda' \\ uu\ldots u & u^*\lambda & \\ & & \bar{u}u\ldots uu^*\lambda. \end{array}$$

Here in the third row the leftmost u (marked $\bar{u}$) of the righthand occurrence of $x[1..c']$ must partially overlap with the rightmost u (marked $\hat{u}$) of the lefthand occurrence of $x[1..c']$ in the first row. We see that therefore a nonempty suffix of length at least

$$|u| - \big(|u^*\lambda| + 1\big) = |u| - |u^*| \geq 1$$

of $\hat{u}$ must coincide with a prefix of $\bar{u}$ of the same length. But this means that $\bar{u}$ is a substring of $\hat{u}u^*$, which implies that u is a nontrivial rotation of itself, as we have seen an impossibility.

Thus the original assumption that the substring $x[1..c]\lambda'$ occurs in x must be false, and so the lemma is proved. $\Box$

We conclude this section by introducing two concepts important for the algorithm of Section 3 together with corresponding lemmas. The first of these ideas is that of a "live" prefix.

Given x, any string of which x is a prefix is called a *right extension* of x. If a prefix u of x can possibly be a cover of some right extension of x, then u is said to be *live with respect to* x; otherwise, u is said to be *dead with respect to* x. For example, if

$$x = abaab,$$

then $u = aba$ is live, since aba covers the right extension xa, while $u = a$ and $u = ab$ can cover no right extension of x, and so are dead.

Since a prefix is identified by its length, and since we speak always of a single given string x, we extend this terminology to positions in x: we say that j is live with respect to i if and only if $x[1, j]$ can possibly be a cover of some right extension of $x[1, i]$. Also, since positions in x correspond to nodes in the cover tree, we extend the terminology further to speak of live and dead nodes in T_γ. Observe that if j is dead with respect to i, it is dead also with respect to $i+1$. Thus, as the string x is scanned from left to right — that is, as i increases — the number of dead positions is monotone nondecreasing.

The first lemma provides a characterization of live positions:

Lemma 2.5 *With respect to every $i \in 1..n$, j' is live if and only if $x[1..j']$ is a cover of some $x[1..j]$, where $j \in i-\beta[i]..i$.*

Proof Suppose first that j' is live with respect to i. Then there exists a minimum-length right extension $y = x[1..i]v$, $0 \le |v| < i$, of $x[1..i]$ such that $x[1..j']$ is a cover of y. Hence $x[1..j' - |v|]$ is a border of $x[1..i]$, and so for some $j \in i - \big(j' - |v|\big)..i$, $x[1..j']$ must cover $x[1..j]$. Since $\beta[i]$ is the length of the longest border of $x[1..i]$, $j' - |v| \le \beta[i]$, and so $j \in i - \beta[i]..i$, as required.

To prove sufficiency, suppose that $x[1..j']$ is a cover of some $x[1..j]$, $j \in i - \beta[i]..i$. Write $x[1..i] = u^r u'$ in normal form, where $r = \lfloor i/|u| \rfloor \ge 1$, u' is a proper prefix of u, and $\beta[i] = |u^{r-1}u'|$. Then u is a prefix of $x[1..j]$:

$$x[1..j] = u^s u''$$

for some $1 \leq s \leq r$ and some proper prefix u'' of u. Two cases now arise:

- If x is a prefix of u, $x[1..j']$ must cover $u^t u''$ for any $t \geq s$; in particular, $x[1..j']$ must cover a right extension $u^{r+1}u''$ of $x[1+i]$, and so j' is live with respect to i.

- If on the other hand u is a prefix of $x[1..j']$, it follows that

  $$x[1..j'] = u^t u'''$$

  for some $t \geq 1$ and some proper prefix u''' of u. Therefore $x[1..j']$ covers any string $u^k u'''$, $k \geq t$, and in particular $u^{r+1}u'''$, again a right extension of $x[i+1]$. Thus in this case also j' is live with respect to i.

□

Since $x[1..j]$ covers itself, Lemma 2.5 tells us that every $j \in i - \beta[i]$ is live with respect to i.

The second main idea to be introduced is that of a "run", already implicit in Theorem 2.3. A *run* $R_{i,h}$ is a *maximal* sequence of $h \geq 1$ positions in the border array β such that $\beta[j+1] = \beta[j] + 1$ for every $j \in i..i+h-1$. Note that the maximality of the run implies that $\beta[i] \leq \beta[i-1]$, for $i - 1 \geq 1$, and $\beta[i+h-1] \geq \beta[i+h]$, for $i + h \leq n$. For example, if we have $\beta = 0011232345645$, there are runs $R_{1,1}$, $R_{2,2}$, $R_{4,3}$, $R_{7,5}$ and $R_{12,2}$.

Observe that at the start position i of every run in the β array, Theorem 2.3 implies that $\gamma[i] = 0$. Observe also as a result of Lemma 2.5 that within a run values i and $\beta[i]$ both increase by 1 together, so that the set of dead nodes remains unchanged; thus the only time at which the number of dead nodes can increase is at the beginning of a run.

As we shall see in Section 3, it is important for our algorithm to characterize dead nodes with respect to the beginning of a run:

Lemma 2.6 *Let i denote a position that starts a run in a border array β. Then $j < i - \beta[i]$ is dead with respect to i if and only if j has no children in T_γ that are live with respect to i.*

Proof Suppose $j < i - \beta[i]$ is dead with respect to i. Then $x[1..j]$ cannot possibly cover any right extension of $x[1..i]$. If j has a child j' in T_γ that is live with respect to i, then $x[1..j']$ is a potential cover of $x[1..k]$, for some $k \geq i$. But since j' is a child of j, it must be true that $x[1..j]$ covers $x[1..j']$ and so also the same $x[1..k]$, a contradiction. Thus necessity is proved.

To prove sufficiency, suppose that $j < i - \beta[i]$ has no child live with respect to i, but that j itself is live with respect to i. Then, by Lemma 2.5, $x[1..j]$ must cover some string $x[1..k]$, $i - \beta[i] \leq k \leq i$. Therefore j has a child k in T_γ that, again by Lemma 2.5, must be live with respect to i, a contradiction. We conclude that j is dead with respect to i. □

As a corollary of this result, we observe that if a node j is live with respect to a node i, then so is its parent $\gamma[j]$ in T_γ.

We state as a final lemma a simple property of $i - \beta[i]$ that is required in the algorithm:

Lemma 2.7 *The function $i - \beta[i]$ is invariant for every i in the same run and monotone nondecreasing in i; in particular, for any position $i > 1$ that starts a new run in β,*

$$(i-1) - \beta[i-1] < i - \beta[i].$$

Proof An immediate consequence of the fact that $\beta[i] \leq \beta[i-1]$ if and only if i starts a run. □

3 COMPUTING THE COVER ARRAY

As mentioned in the introduction, the algorithm performs two main tasks as the string x and border array β are processed from left to right, for each $i = 1, 2, \ldots, n$:

- compute $\gamma[i]$ and add i as a new child of $\gamma[i]$ in the cover tree T_γ — recall that by definition $\gamma[i]$ is just the parent of i in T_γ;

- for every i that marks the start of a run in β, compute the nodes in T_γ that are dead with respect to i.

In addition to $\beta[1..n]$ and $\gamma[1..n]$, the algorithm also uses the following arrays:

- *dead*$[0..n-1]$: by Lemma 2.6, *dead*$[j]$ = *true* if and only if j has no children in T_γ that are live with respect to the current value of i *and* $j < i - \beta[i]$ (as we shall see, the root node 0 in T_γ is always live);

- *livechildren*$[0..n]$: *livechildren*$[i] = k$ if and only if node i in the cover tree T_γ has exactly k live children.
- *largestlive*$[0..n]$: *largestlive*$[i] = j$ if and only if j is the largest live ancestor of i in the cover tree T_γ (of course i is always live with respect to itself).

The algorithm is initialized by placing node 0 in the cover tree T_γ, setting every position in *dead* to *false*, every position in *livechildren* to 0, and *largestlive*$[i]$ to i for every $i \in 1..n$. Then the procedure *AllCovers* outlined below is executed on the current i^{th} position in x, $i = 1, 2, \ldots, n$. *AllCovers* has just three steps: Step 1 ensures that the *largestlive* array is kept up to date, Step 2 attaches i to its proper parent in T_γ, and Step 3 ensures that the *dead* array is updated at the start of each new run.

```
1. {If $\beta[i]$ in $T_\gamma$ is dead, $\beta[i]$ should have the same largest live ancestor as its parent does.}
   If $dead\big[\beta[i]\big] = true$, then $largestlive\big[\beta[i]\big] \leftarrow largestlive\Big[\gamma\big[\beta[i]\big]\Big]$.
2. {Compute $\gamma[i]$: if $\beta[i]$ is live, set $\gamma[i] \leftarrow \beta[i]$; otherwise, since every cover of $x[1..i]$ must cover $x\big[1..\beta[i]\big]$, set $\gamma[i]$ equal to the largest live ancestor (possibly 0) of $\beta[i]$.}
   $\gamma[i] \leftarrow largestlive\big[\beta[i]\big]$; increment $livechildren\big[\gamma[i]\big]$.
3. {Identify all the nodes in $T_\gamma$ that have become dead as a result of starting the new run.}
   If $i > 1$ starts a new run in the border array $\beta$:
   3.1 Compute $c_1 \leftarrow i - \beta[i]$, $c_2 \leftarrow (i-1) - \beta[i-1]$ (Lemma 2.7).
   3.2 For every $j = c_1 - 1, c_1 - 2, ..., c_2$ (Lemma 2.6):
         * if not $dead[j]$ and $j$ has no live children in $T_\gamma$, then
               $dead[j] \leftarrow true$; decrement $livechildren\big[\gamma[j]\big]$;
         * (recursively) if $j$ has just been set dead and $\gamma[j]$ has no live children in $T_\gamma$, then
               $dead\big[\gamma[j]\big] \leftarrow true$; decrement $livechildren\Big[\gamma\big[\gamma[j]\big]\Big]$.
```

We claim that

Theorem 3.1 *Procedure AllCovers computes the cover array $\gamma[1..n]$ correctly.*

Proof We consider the steps separately, beginning with Step 3.

Let $1 = i_1 < i_2 < \cdots < i_k \leq n$ denote the positions in β at which runs start. In Step 3 each position $j \in i_{h-1}..i_h - 1$, $h = 2, 3, \ldots, k-1$, is tested to determine whether or not it should be set dead with respect to i_{h+1} — by Lemma 2.5, each such position was live with respect to i_h. By Lemma 2.6, the parent $\gamma[j]$ of any position j that is set dead must be inspected: $\gamma[j]$ must previously have been live, but it will now be dead if and only if it has no live children. Thus, Step 3 must recursively examine the parent of *any* position that has been set dead, until an ancestor is found that has at least one live child, and that therefore remains alive. Note that since i_{h+1} is always live with respect to itself, there must by Lemma 2.6 exist a path containing only live nodes that leads from i_{h+1} to the root. Thus the root is always live, and Step 3 will always terminate at the first live node along the path from j to the root.

Note that the positions $j \in i_{h-1}..i_h - 1$ are considered in the reverse order $i_h - 1, i_h - 2, \ldots, i_{h-1}$ in order to take account of the possibility that j may be set dead and that moreover $\gamma[j] \in i_{h-1}..i_h - 1$ also. It is for this same reason that, in order to avoid redundant processing of the same path to the root, it is necessary to check in Step 3.2 that **not** $dead[j]$: j could have been set dead because it is the parent of a larger node, already set dead, in $i_{h-1}..i_h - 1$.

Since $livechildren\big[\gamma[j]\big]$ is always decremented for every node j that is set dead, we conclude that Step 3 deals correctly with the task of setting nodes dead at the start of each new run.

Step 2 is a straightforward update of T_γ based on the current position i. Its correctness depends entirely on the correct update of $largestlive\big[\beta[i]\big]$ in Step 1.

Step 1 will be executed only when $\beta[i]$ has been set dead during one of the previous executions of Step 3.2. As noted in the proof of Step 3, if some j is to be set dead with respect to i_{h+1}, it must have been live with respect to i_h, hence by Lemma 2.5 live with respect to $i_{h+1} - 1$. Therefore $x[1..j]$ would be a *potential* cover of $x[1..i_{h+1}]$; thus it would cover some right extension of $x\big[1..\beta[i_{h+1}]\big]$, which is a suffix of $x[1..i_{h+1}]$. Hence the following fact holds: at the time that any position j is set dead, it must be true that j is greater than $\beta[i_{h+1}]$, where i_{h+1} is the current value used in Step 3.1.

Since by Lemma 2.1 values in the β array can increase by at most one from a given position to the next, it follows that every such dead position j must later become a value in the β array — that is, $j = \beta[i']$ for some $i' > i_{h+1}$ — in order for Step 1 to be executed. In particular, the values j must be processed in Step 1 in ascending order of magnitude; that is, in descending order in the cover tree T_γ. This means that Step 1 will pass correct values of the largest live ancestor from parent to child. $\Box$

Now consider the time required by *AllCovers*. Each of the steps except possibly Step 3.2 requires only constant time. Then *AllCovers* requires $\Theta(n)$ time plus the total time used within Step 3.2. To estimate this total time, observe that the time required for *each* execution of Step 3.2 is proportional to

$$\max\{(c_1 - c_2), \text{no. of nodes set dead}\}.$$

Since the sum of $c_1 - c_2$ over all runs is at most $n-1$ and since each of the n nodes may be set dead at most once, it follows that the total time for Step 3.2 is $O(2n)$. Hence

Theorem 3.2 *Procedure AllCovers requires $\Theta(n)$ time and $\Theta(n)$ space for its execution.* $\Box$

AllCovers is an optimal **on-line** algorithm: it computes γ, thus making available all the covers of every prefix of x, in $\Theta(n)$ time and space. Note that since the calculation of β is also on-line, *AllCovers* can easily be modified to simultaneously compute on-line both the border and the cover arrays.

The reader will find it instructive to follow the algorithm as it applies to the example given in Figure 1. Note particularly that, as a result of the fact that $\beta[22] = 2$, the following nodes are all set dead: $5, 6, 9, 10, 11, 13-19$.

References

[1] Alberto Apostolico & Andrzej Ehrenfeucht, *Efficient Detection of Quasiperiodicities in Strings*, Tech. Report No. 90.5, The Leonardo Fibonacci Institute, Trento, Italy (1990).

[2] Alberto Apostolico, Martin Farach & Costas S. Iliopoulos, **Optimal superprimitivity testing for strings**, *IPL* 39-1 (1991) 17-20.

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

[4] Alberto Apostolico & F. P. Preparata, **Optimal off-line detection of repetitions in a string**, *TCS* 22 (1983) 297-315.

[5] O. Berkman, Costas S. Iliopoulos & Kunsoo Park, **The subtree max gap problem with application to parallel string covering**, *Information & Computation* 123-1 (1995) 127-137.

[6] D. Breslauer, **An on-line string superprimitivity test**, *IPL* 44-6 (1992) 345-347.

[7] D. Breslauer, **Testing string superprimitivity in parallel**, *IPL* 49-5 (1994) 235-241.

[8] Maxime Crochemore, **An optimal algorithm for finding the repetitions in a word**, *IPL* 12-5 (1981) 244-248.

[9] Ming Gu, Martin Farach & Richard Beigel, **An efficient algorithm for dynamic text indexing**, *Proc. Fifth Annual ACM-SIAM Symp. on Discrete Algorithms* (1994) 697-704.

[10] Costas S. Iliopoulos & Laurent Mouchard, **Fast local covers**, preprint (1999).

[11] Costas S. Iliopoulos & Kunsoo Park, **An optimal $O(n \log n)$-time algorithm for parallel superprimitivity testing**, *J. Korea Information Sci. Soc.* 21-8 (1994) 1400-1404.

[12] Costas S. Iliopoulos & W. F. Smyth, **An on-line algorithm for computing a minimum set of k-covers of a string**, preprint.

[13] M. Lothaire, *Combinatorics on Words*, Addison-Wesley (1982).

[14] Michael G. Main & Richard J. Lorentz, **An $O(n \log n)$ algorithm for finding all repetitions in a string**, *J. Algs.* 5 (1984) 422-432.

[15] Dennis Moore & W. F. Smyth, **An optimal algorithm to compute all the covers of a string**, *IPL* 50 (1994) 239-246.

[16] Dennis Moore & W. F. Smyth, **Correction to: An optimal algorithm to compute all the covers of a string**, *IPL* 54 (1995) 101-103.

[17] Axel Thue, **Über unendliche zeichenreihen**, *Norske Vid. Selsk. Skr. I, Mat. Nat. Kl. Christiana* 7 (1906) 1-22.

ACKNOWLEDGEMENTS

The work of the second author was supported in part by Grant No. A8180 of the Natural Sciences & Engineering Research Council of Canada and by Grant No. GO-12278 of the Canadian Genome Analysis & Technology Agency.