Correction to

AN OPTIMAL ALGORITHM TO COMPUTE ALL THE COVERS OF A STRING

Dennis Moore

School of Computing
Curtin University of Technology

W. F. Smyth

Department of Computer Science & Systems
McMaster University

School of Computing
Curtin University of Technology

ABSTRACT

This note corrects an error in a paper recently published in this journal (**An optimal algorithm to compute all the covers of a string**, *IPL 50-5* (1994) 239-246). The correction consists primarily of a new subalgorithm which is called by COMPUTE_COVERS, the main algorithm presented in the paper referenced. It turns out that the new subalgorithm is itself sufficient to solve the original problem — that is, to compute all the covers of a given string in time linear in the string length — and so it is presented here as a self-contained algorithm in its own right.

1 INTRODUCTION

For notation and terminology see [1]. The error in [1] relates to case (c) of Theorem 2.1, where it is shown that the proper covers of a given string $x = v^*vv^*$ must in fact be covers of v^* which also cover x. (Here $v^*v = x[1..k_1]$ and $v^* = x[1..k_2]$ are substrings identified when x is expressed in *normal form*.) Thus the problem of computing the proper covers of x is reduced to the problem of computing the covers u of v^*, provided that it can be efficiently checked that these covers u are also covers of x. On page 244 of [1] the following statement is made:

> 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 $f[i] < |u|$.

(Here $f[i]$ is an element of the "failure array" $f = f[1..n]$ and specifies the length of the longest border of $f[1..i]$.) As pointed out in [2], this statement is incorrect. In order to formulate a correct "if and only if" condition, we first introduce a definition: given nonnegative integers i and h, we say that i *reduces to* h iff there exists a positive integer j such that $f^j[i] = h$ (where f^j denotes j compositions of f). The following result then leads to a correct statement of the condition:

Lemma 1.1 Let h and r denote integers satisfying $1 \leq h \leq r \leq n$, and suppose that $u = x[1..h]$ is a cover of $x[1..r]$. Then u is a cover of x if and

Typeset by $\mathcal{AMS}$-TEX

only if there exist at most $h-1$ consecutive positions of $x[r+1..n]$ which do not reduce to h.

Proof If u is a cover of x, then the substring $x[1..h]$ begins in at least every h^{th} position of x, and every such occurrence terminates in the letter $x[h]$ at some position i, where for $i \geq h$, it follows that i reduces to h.

Now suppose that at least every h^{th} position of $x[r+1..n]$ reduces to h. Observe that since u is a cover of $x[1..r]$, it must be true that at least every h^{th} position of $x[h+1..r]$ reduces to h. Thus at least every h^{th} position of $x[h+1..n]$ reduces to h, and each of these positions marks the end of an occurrence of u. Hence u covers x. □

In view of Lemma 1.1, a correct necessary and sufficient condition for a cover u of v^* to also be a cover of $x = v^*vv^*$ is that there should exist at most $|u|-1$ consecutive positions i of vv^* such that i does not reduce to $|u|$. Consider then an integer k' which is constrained to be the length of a border of vv^*. It turns out that the stated condition can be efficiently implemented by performing (at most once) a subalgorithm which computes the greatest of these integers k' such that at most $k'-1$ consecutive positions of vv^* do not reduce to k'. Then having found such a greatest value of k', it suffices, in order to check that a cover u of v^* is also a cover of x, to compare $|u|$ with k', a constant time operation: u will be a cover of x if and only if $|u| \leq k'$.

It turns out further that the subalgorithm which computes the greatest value of k' is merely a special case of an algorithm which considers in turn the borders of x itself, deciding for each one whether or not it is in fact a cover of x. It is this slightly more general algorithm that is described in the next section.

2 DECIDING WHETHER A BORDER OF x IS A COVER

We suppose from now on that x is an arbitrary string of length n with exactly m borders $x[1..b_1], x[1..b_2], \ldots, x[1..b_m]$, where for $j = 1, 2, \ldots, m$, $b_j = f^j[n]$, while $f^{m+1}[n] = f[b_m] = 0$. Note that $b_1 > b_2 > \cdots > b_m$; to exclude trivial cases, we assume without loss of generality that $b_m > i_0$, where i_0 is the greatest integer for which $f[i_0] = 0$. Our task is to determine whether or not each $x[1..b_j]$ is a cover of x. Applying Lemma 1.1 with $h = r = b_j$, we see that this task is equivalent to determining whether or not it is true that there exist at most $b_j - 1$ consecutive positions of $x[b_j+1..n]$ which do not reduce to b_j.

The main idea used in our algorithm is that of a "border tree", which we now define. A *border tree* B_x is a rooted tree 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, $i = 1, 2, \ldots, n$, is the node with label $f[i]$. It is clear that B_x is in fact a tree, and we observe that the descendants of the node labelled i in B_x are exactly those nodes whose labels reduce to i. Thus by arranging these descendant nodes in ascending label sequence, we can easily determine whether any difference between adjacent labels in the sequence exceeds i or not; if not, and if $x[1..i]$ is a border of x, then by Lemma 1.1 we are entitled to conclude that $x[1..i]$ is also a cover of x.

The algorithm first considers the shortest border $x[1..b_m]$: a collection of all the nodes in the subtree of B_x rooted at b_m is formed, and these nodes are sorted, using a binsort, into ascending label sequence. These sorted labels are then added into a doubly-linked list L_m whose initial element is a dummy element with label 0; as each label is added to the list, the difference between the current and the preceding label is computed, so that the quantity MAX_GAP, the maximum difference between adjacent labels, can be maintained. When L_m has been fully updated, $x[1..b_m]$ will be a cover if and only if $b_m \geq$ MAX_GAP.

The algorithm now considers each b_j in turn, $j = m-1, m-2, \ldots, 1$. For each j, the labels contained in the subtree rooted at b_{j+1} but NOT in the subtree rooted at b_j are deleted from L_{j+1}, yielding L_j. (Observe that b_{j+1} is necessarily the parent of b_j in the border tree because $b_{j+1} = f[b_j]$.) As each label is deleted, the preceding and following labels are inspected, and the difference between these labels is computed; if this difference exceeds MAX_GAP, then MAX_GAP is updated with the computed difference. When the formation of L_j is complete, $x[1..b_j]$ will again be a cover if and only if $b_j \geq$ MAX_GAP.

In order to be able to handle the deletions from L_{j+1} corresponding to b_j in time proportional to the number of nodes deleted, a strategy needs to be implemented which identifies those nodes. One way to accomplish this is to introduce at each position in the subtree rooted at b_m a pointer to the corresponding position in L_m; then the deletions can be effected by traversing the appropriate subtrees of b_{j+1}.

The algorithm is summarized below, with notes indicating the time required for each step.

(1) Compute the failure array $f = f[1..n]$ in time $\Theta(n)$.

(2) Compute the border tree B_x in time $\Theta(n)$. In practice B_x will be implemented as an equivalent binary tree. Also the construction of B_x will usually be limited only to those nodes occurring in the subtree rooted at b_m.

(3) Perform the binsort of the labels in the subtree of B_x rooted at b_m. This requires $\Theta(n)$ time. Then, in $O(n)$ time, construct the doubly-linked list L_m and compute MAX_GAP. If $b_m \geq$ MAX_GAP, then output b_m.

(4) For each $j = m-1, m-2, \ldots, 1$, compute L_j from L_{j+1} in time proportional to the number of nodes deleted. At the same time, recompute MAX_GAP; if $b_j \geq$ MAX_GAP, then output b_j. The time required over all values of j is proportional to the number of nodes in L_m, hence $O(n)$.

This algorithm computes all the covers of $x = x[1..n]$ in $\Theta(n)$ time and $\Theta(n)$ space. Observe that, with trivial modifications, the same algorithm can be used to compute all the covers of any prefix $x[1..i]$ in $\Theta(i)$ time and $\Theta(i)$ space.

As an example of the algorithm, consider the string

$$x = (ab)^6(aab)^2aba$$

with failure array

$$f = (0,0,1,2,3,4,5,6,7,8,9,10,11,1,2,3,1,2,3,4,5),$$

which yields the border tree shown in Figure 2.1. Since $f[21] = 5$ and $f[5] = 3$, we find $m = 3$ and $b_3 = 3$, $b_2 = 5$, $b_1 = 21$. The initial list $L_3 = 0-3-5-7-9-11-13-16-19-21$, from which we compute MAX_GAP $= 3$ and so accept $x[1..b_3] = aba$ as a cover. For $j = 2$ we compute $L_2 = 0-5-7-9-11-13-21$ and MAX_GAP $= 8$, and thus reject $x[1..b_2] = ababa$ as a cover. Finally, we compute $L_1 = 0-21$ and accept $x = x[1..b_1]$ as a cover.

REFERENCES

[**1**] Dennis Moore & W. F. Smyth, **An optimal algorithm to compute all the covers of a string**, *IPL 50-5* (1994) 239-246.

[**2**] Kunsoo Park, private communication.

ACKNOWLEDGEMENT

The work of the second author was supported in part by Grant No. A8180 of the Natural Sciences & Engineering Research Council of Canada. The authors gratefully acknowledge the contributions of Kunsoo Park, who found the error in the originally published algorithm, and of Costas Iliopoulos, for enlightening discussions. They also express their thanks to Dianne Miller for her C implementation of the algorithm presented in Section 2.