Algorithmica (1996) 16: 288–297

Algorithmica

Covering a String

C. S. Iliopoulos,[1] D. W. G. Moore,[2] and K. Park[3]

Abstract. We consider the problem of finding the repetitive structures of a given string x. The period u of the string x grasps the repetitiveness of x, since x is a prefix of a string constructed by concatenations of u. We generalize the concept of repetitiveness as follows: A string w *covers* a string x if there is a superstring of x which is constructed by concatenations and superpositions of w. A substring w of x is called a *seed* of x if w covers x. We present an $O(n \log n)$-time algorithm for finding all the seeds of a given string of length n.

Key Words. Combinatorial algorithms on words, String algorithms, Periodicity of strings, Covering of strings, Partitioning.

1. Introduction. Regularities in strings arise in many areas of science: combinatorics, coding and automata theory, molecular biology, formal language theory, system theory, etc. Here we study string problems focused on finding the repetitive structures of a given string x. A typical regularity, the period u of the string x, grasps the repetitiveness of x, since x is a prefix of a string constructed by concatenations of u. We consider a problem derived by generalizing this concept of repetitiveness by allowing overlaps between the repeated segments.

Apostolico *et al.* [3] considered the *Aligned String Covering* (ASC) problem: Given a string x, find a substring w of x such that x can be constructed by concatenations and superpositions of w; the string w is said to provide an *aligned cover* for x. The shortest such w ($\neq x$) is called the *quasi-period* of x, and x is said to be *quasi-periodic* if x has a quasi-period. For example, if $x = abaababababaaba$, then x is quasi-periodic with the quasi-period $w = aba$. A linear-time algorithm for computing the quasi-period was given in [3]. Breslauer [7] presented a linear-time on-line algorithm for the same problem. Moreover, Apostolico and Ehrenfeucht [2] introduced *maximal quasi-periodic* substrings of a string x, and presented an $O(n \log^2 n)$-time algorithm for computing all maximal quasi-periodic substrings of x.

We focus on the *General String Covering* (GSC) problem. We say that a string w *covers* a string x if a superstring of x exists which is constructed by concatenations and superpositions of w. For example, $abca$ covers $abcabcaabc$. A substring w of a string

[1] Department of Computer Science, King's College London, Strand, London, England, and School of Computing, Curtin University, Perth, WA, Australia. csi@dcs.kcl.ac.uk. Partially supported by SERC Grants GR/F 00898 and GR/J 17844, NATO Grant CRG 900293, ESPRIT BRA Grant 7131 for ALCOM II, and MRC Grant G 9115730.

[2] School of Computing, Curtin University, Perth, WA, Australia. moore@marsh.cs.curtin.edu.au.

[3] Department of Computer Engineering, Seoul National University, Seoul 151-742, Korea. kpark@theory.snu.ac.kr. Partially supported by MRC Grant G 9115730 and S.N.U. Posco Research Fund 94-15-1112.

Received November 18, 1993; revised November 5, 1994. Communicated by J.-D. Boissonnat.

x is called a *seed* of x if w covers x. The GSC problem is as follows: Given a string x of length n, compute all the seeds of x. Note that there may be more than one shortest seed (e.g., for $abababa$, both ab and ba are the shortest seeds).

The aligned string covering problem considered in [3] is restricted in the following sense. The computational focus of the ASC problem is on finding the quasi-period of x rather than finding all possible strings which provide aligned covers for x. More importantly, the first and the last occurrences of the quasi-period w in x must align exactly with the given string x. Therefore, the period of x may not provide an aligned cover for x, though a periodic string is quasi-periodic. Consider the string $abcabcabca$, which is quasi-periodic with the quasi-period $abca$. However, the period abc does not provide an aligned cover for the string.

We present an $O(n \log n)$-time algorithm for finding all the seeds of a given string of length n. Since the possible number of seeds of a string of length n can be as large as $\Theta(n^2)$, this is achieved by reporting a group of seeds in one step.

In Section 2 we classify the seeds of a given string into two kinds: easy seeds and hard seeds. Then we describe how to find all easy seeds in $O(n)$ time. In Sections 3 and 4 we describe how to find all hard seeds in $O(n \log n)$ time. In Section 5 we mention open problems related to the general string covering problem.

2. Preliminaries. A *string* is a sequence of zero or more symbols from an alphabet Σ. The set of all strings over the alphabet Σ is denoted by Σ^*. A string x of length n is represented by $x_1 \cdots x_n$, where $x_i \in \Sigma$ for $1 \le i \le n$. A string w is a *substring* of x if $x = uwv$ for $u, v \in \Sigma^*$; x is a *superstring* of w. A string w is a *prefix* of x if $x = wu$ for $u \in \Sigma^*$. Similarly, w is a *suffix* of x if $x = uw$ for $u \in \Sigma^*$.

The string xy is a *concatenation* of two strings x and y. The concatenations of k copies of x is denoted by x^k. For two strings $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_m$ such that $x_{n-i+1} \cdots x_n = y_1 \cdots y_i$ for some $i \ge 1$, the string $x_1 \cdots x_n y_{i+1} \cdots y_m$ is a *superposition of x and y with i overlaps*. A string $w = w_1 \cdots w_n$ is a *cyclic rotation* of $x = x_1 \cdots x_n$ if $w_1 \cdots w_n = x_i \cdots x_n x_1 \cdots x_{i-1}$ for some $1 \le i \le n$ (for $i = 1$, $w = x$). For a substring w of x, uwv for $u, v \in \Sigma^*$ is an *extension* of w in x if uwv is a substring of x; wv for $v \in \Sigma^*$ is a *right extension* of w in x if wv is a substring of x; uw for $u \in \Sigma^*$ is a *left extension* of w in x if uw is a substring of x.

A string u is a *period* of x if x is a prefix of u^k for some k, or equivalently if x is a prefix of ux. The shortest period of x is *the period* of a string x. A string w *a-covers* ("a" stands for alignment) a string x if x can be constructed by concatenations and superpositions of w. A string w *covers* a string x if a superstring z of x exists which is constructed by concatenations and superpositions of w. Such a string z is called a *cover* of x by w, and the shortest cover is called *the cover* of x by w. A *seed* of a string x is a substring of x that covers x.

For example, for $x = ababaababa$, the strings aba and $ababa$ a-cover (but also cover) x; the strings $baaba$ and $baabab$ cover (but do not a-cover) x. The strings aba, $ababa$, $baaba$ and $baabab$ are seeds of x. The cover of x by aba is x itself, the cover of x by $baaba$ is $baxaba$ and the cover of x by $baabab$ is $baxabab$.

In the definition of the seed, we do not consider strings that are longer than the given string x, because we are interested in the repetitive structures of x. We also restrict ourselves to substrings of x, because any other string w that covers x can be easily derived from a substring of x as Theorem 1 below shows.

LEMMA 1. *Let w be a seed of x. If w covers x only by concatenations, then:*

(1) *All the cyclic rotations of w cover x by concatenations.*
(2) *All the extensions of w in x cover x.*

PROOF. (1) Since w covers x by concatenations, there is a cover w^k of x by w. Let $\hat{w}$ be a cyclic rotation of w. Since w^k is a substring of $\hat{w}^{k+2}$, $\hat{w}^{k+2}$ is a cover of x and therefore $\hat{w}$ covers x by concatenations.

(2) Let $\hat{w} = uwv$ be an extension of w in x. Since w covers x by concatenations, there is a cover w^k of x by w, and u and v are a suffix and a prefix of w, respectively. Thus uw^kv is constructed by superpositions of k copies of $\hat{w}$. Since uw^kv is a cover of x by $\hat{w}$, $\hat{w}$ covers x. □

THEOREM 1. *Let y be a string such that $|y| \leq |x|$ and y is not a substring of x. If y covers x, then there is a seed of x that is either a substring of y or a cyclic rotation of y or a cyclic rotation of a substring of y.*

PROOF. Let $m = |y|$ and $n = |x|$. Since $m \leq n$ and y is not a substring of x, y covers x with exactly two copies. Let z be the cover of x by y, and let i be the integer such that x is a prefix of $z_i \cdots z_{|z|}$.

1. $z = y^2$: Let $w = y_i \cdots y_m y_1 \cdots y_{i-1}$, which is a cyclic rotation of y. Since $|w| = m$, w is a prefix of x. Since w^2 is a cover of x, w is a seed of x. By Lemma 1(1), y is derived from w.
2. $z = yy_j \cdots y_m$ for $j > 1$. There are two cases:
   2.1. $i \leq j$: Let $w = y_i \cdots y_{m-j+i}$, which is a substring of y. Since $|w| = m-j+1 < m$, w is a prefix of x. Since w^3 is a cover of x, w is a seed of x. By Lemma 1(2), y is derived from w.
   2.2. $i > j$: Let $w = y_i \cdots y_m y_j \cdots y_{i-1}$, which is a cyclic rotation of $y_1 \cdots y_{m-j+1}$. Since $|w| = m - j + 1 < m$, w is a prefix of x. Since w^2 is a cover of x, w is a seed of x. By Lemma 1, y is derived from w. □

For a string of length n there may be $\Theta(n^2)$ different substrings, each of which can possibly be a seed. In fact, the string $(a_1a_2 \cdots a_m)^4$, where $a_i \neq a_j$ for $i \neq j$, has $\Theta(m^2)$ seeds. To get $o(n^2)$ operations, our algorithm reports a group of seeds in one step. Each seed w of x is reported by a pair (i, j), where i and j are the start position and the end position, respectively, of an occurrence of w in x. If $(i, j), (i, j+1), \ldots, (i, j+k)$ are seeds, we report them by a triple (i, j, k). If $(i, j), (i-1, j), \ldots, (i-k, j)$ are seeds, we report them by a triple $(i, j, -k)$. By this representation, the output size of our algorithm will be $O(n)$.

We classify the seeds of x into two kinds: A seed w is an *easy seed* if there is a substring of w which covers x only by concatenations; w is a *hard seed* otherwise. For example, for $x = (abbab)^3abb$, the strings $abbab$, $babab$ cover x by concatenations and thus are easy seeds. The strings *babbab*, *bababba* are also easy seeds of x, having *abbab* and *babab* as substrings respectively which cover x by concatenations. However, the string *bab* is a hard seed of x. Let $u = x_1 \cdots x_p$ be the period of x. It is easy to see that u covers x by concatenations. The following lemmas characterize easy seeds of x.

LEMMA 2. *A seed w is an easy seed if and only if $|w| \geq p$.*

PROOF. (if) Let w be a seed such that $|w| \geq p$. Since w is a substring of x and $|w| \geq p$, w contains a cyclic rotation $\hat{u}$ of u as a substring. Since $\hat{u}$ covers x by concatenations by Lemma 1(1), w is an easy seed.

(only if) Let w be an easy seed. Suppose that $|w| < p$. Since w is an easy seed, there is a substring $\hat{w}$ of w that covers x by concatenations; i.e., x is a substring of w^k. Let w' be the cyclic rotation of $\hat{w}$ which is a prefix of x. Then w' covers x by concatenations by Lemma 1(1), and therefore w' is a period of x. Since the length of w' is less than p, we have a contradiction. Therefore, $|w| \geq p$. □

LEMMA 3. *A substring w of x is an easy seed if and only if w is a right extension of a cyclic rotation of u.*

PROOF. (if) Let w be a right extension of a cyclic rotation $\hat{u}$ of u. Since $\hat{u}$ covers x by concatenations by Lemma 1(1), w is an easy seed.

(only if) Let w be an easy seed. By Lemma 2, $|w| \geq p$. Let $\hat{w} = w_1 \cdots w_p$. Since $\hat{w}$ is a substring of x whose length is p, it is a cyclic rotation of u. Thus w is a right extension of a cyclic rotation of $\hat{x}$. □

We first find all the easy seeds of x. The period length p is computed in $O(n)$ time by the preprocessing of the Knuth–Morris–Pratt algorithm [11] for string matching. By Lemma 3, all cyclic rotations of the period and their right extensions are seeds of x. Therefore, all easy seeds can be reported in $O(p)$ time. Including the preprocessing, we find easy seeds in $O(n)$ time. In the following sections we find all the hard seeds in $O(n \log n)$ time.

3. Finding Candidate-Sets.

For the computation of hard seeds we need additionally the following definitions. A substring w of the given string x is a *candidate* for a hard seed if there is a substring x' of $x = ux'v$ such that w a-covers x' and $|u|, |v| < |w|$. See Figure 1. For maximal such x' we call u (resp. v) the *head* (resp. *tail*) of x with respect

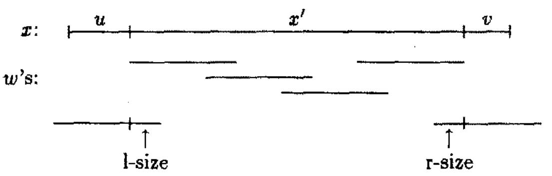

Fig. 1. A candidate w of the given string x.

to w. In order for the candidate w to be a seed, the head u and the tail v must also be covered by w. If w covers the head u, then among all such coverings consider the copy of w whose overlap with x is maximal. We define *l-size* to be the length of the overlap between x' and the above copy of w. Similarly, if w covers the tail v, then consider the copy of w whose overlap with x is maximal. We define *r-size* to be the length of the overlap between x' and the above copy of w. See Figure 1. For example, consider the following string:

$$x = cababacababacabacabacabac,$$

then the string $w = abacab$ is a candidate for hard seed of x, since w a-covers x', where

$$x = cabx'ac \quad \text{with} \quad x' = abacababacabacabacab.$$

Moreover, cab (resp. ac) is the head (resp. tail) of x with respect of w; the l-size is 0 and the r-size is 2. We divide hard seeds into two types:

(1) A hard seed is a type-A seed if its l-size is larger than or equal to its r-size.
(2) A hard seed is a type-B seed if its l-size is smaller than its r-size.

We describe our algorithm for finding all type-A seeds. The algorithm for type-B seeds is analogous, and is discussed at the end of Section 4.

For each substring w of x, the *start-set* of w is the set of start positions of all occurrences of w in x. For each start-set, we maintain its elements in ascending order. An *equi-set* is a set of substrings of x whose start-sets are the same. Note that a start-set is associated with an equi-set and vice versa. Although there are $O(n^2)$ substrings of x, there are only $O(n)$ distinct start-sets.

LEMMA 4 [6], [9]. *For a string of length n there are $O(n)$ distinct start-sets* (*i.e.*, $O(n)$ *equi-sets*).

A *candidate-set* is a set S of candidates w_i, $f \leq i \leq g$, such that:

(i) $|w_i| = i$.
(ii) All w_i's have the same start-set.

Thus a candidate-set is a subset of an equi-set. In Figure 2 equi-sets S_1 and S_2 do not produce candidate-sets because aba and ba are not candidates. S_3 produces a candidate-set $\{baaaba\}$; S_4 a candidate-set $\{ababaaaba\}$; and S_5 a candidate-set $\{abaaab, abaaaba\}$.

In this section we find all the candidate-sets of the given string x. Since the length of a hard seed is less than p by Lemma 2, we find candidates whose length is less than p. We first find all start-sets, then find equi-sets associated with the start-sets, and finally find at most one candidate-set from each equi-set. By Lemma 4 there will be $O(n)$ candidate-sets.

To find the start-sets, we define equivalence relations E_l for $1 \leq l < n$. E_l is defined on the positions $\{1, 2, \ldots, n-l+1\}$ of x: $i\, E_l\, j$ if $x_i \cdots x_{i+l-1} = x_j \cdots x_{j+l-1}$. Now we maintain equivalence classes for each E_l. Note that the start-set of a substring of length l is an equivalence class of E_l. If a start-set A is an equivalence class of $E_l, \ldots, E_{l'}$, the

$$\begin{array}{l} \ {\scriptstyle 1}\quad\ \ {\scriptstyle 5}\qquad\quad {\scriptstyle 10}\qquad\quad {\scriptstyle 15}\qquad\quad {\scriptstyle 20}\qquad {\scriptstyle 24} \\ x = a\,a\,b\,a\,b\,a\,a\,a\,b\,a\,b\,a\,a\,a\,b\,a\,a\,a\,b\,a\,a\,a\,b\,a \end{array}$$

substring w of x	start-set of w	equi-set
aba	$\{2,4,8,10,14,18,22\}$	$S_1 = \{aba\}$
ba	$\{3,5,9,11,15,19,23\}$	$S_2 = \{ba\}$
baa	$\{5,11,15,19\}$	$S_3 = \{baa, baaa, baaab, baaaba\}$
$baaa$	$\{5,11,15,19\}$	S_3
$baaab$	$\{5,11,15,19\}$	S_3
$baaaba$	$\{5,11,15,19\}$	S_3
$abab$	$\{2,8\}$	$S_4 = \{abab, ababa, \ldots, ababaaaba\}$
$ababaaaba$	$\{2,8\}$	S_4
$abaa$	$\{4,10,14,18\}$	$S_5 = \{abaa, abaaa, abaaab, abaaaba\}$
$abaaaba$	$\{4,10,14,18\}$	S_5

Fig. 2. Start-sets and equi-sets.

equi-set associated with A is the set of strings of length l to l' whose start positions are the elements of A.

We now describe how to find all the start-sets. It is easy to see that E_{l+1} is a refinement of E_l, excluding the position $n-l+1$. The equivalence classes of E_1 can be computed by scanning x in $O(n \log n)$ time when the alphabet is general. Then we compute E_2, $E_3, \ldots$ successively until all classes are singleton sets or $l = p$. At stage l of the refinements we compute E_{l+1} from E_l. The refinement is based on:

$$i\,E_{l+1}\,j \qquad \text{if and only if} \quad i\,E_l\,j \text{ and } (i+1)E_l(j+1).$$

That is, i and j in an equivalence class of E_l belong to the same equivalence class of E_{l+1} if and only if $(i+1)E_l(j+1)$. An easy solution is:

(1) Take each class C of E_l.
(2) Partition C so that $i, j \in C$ go to the same class of E_{l+1} if and only if $(i+1)E_l(j+1)$.

This method leads to $O(n^2)$ time, since each refinement requires $O(n)$ time and there can be $O(n)$ stages of refinements.

We do the refinement more efficiently as follows:

(1) Take a class C of E_l.
(2) Instead of partitioning C, we partition with respect to C those classes D of E_l which has at least one i such that $i+1 \in C$, and one j such that $j+1 \notin C$. That is, each D is partitioned into classes $\{i \in D \mid i+1 \in C\}$ and $\{i \in D \mid i+1 \notin C\}$.

Note that at the end of stage l, for each class D of E_{l+1} there is one class A of E_l such that, for all $i \in D$, $i+1 \in A$. This fact can be more easily observed in terms of strings: if aw for $a \in \Sigma$ and $w \in \Sigma^+$ is the string whose start-set is D, then w is the string whose start-set is A. We call D a *preimage class* of A.

(i) If a class A of E_l is not split at stage l, we need not partition the preimage classes of A with respect to A at stage $l+1$.
(ii) If A is split into $C_1, \ldots, C_r$ at stage l, we need to partition the preimage classes of A at stage $l+1$. For a preimage class D, let $D_s = \{i \in D \mid i+1 \in C_s\}$ for $1 \le s \le r$.

```
procedure PARTITION
  compute E_1;
  SMALL ← all classes of E_1;
  l ← 1;
  while l < p and there is a non-singleton class of E_l do
    copy classes in SMALL into QUEUE;
    empty SMALL;
    l ← l + 1;
    while QUEUE not empty do
      extract a class C from QUEUE;
      partition with respect to C;
      for each split class D, maintain its new subclasses;
    end do
    for each split class D (into r subclasses) do
      put r − 1 small subclasses of D into SMALL;
    end do
  end do
end
```

Fig. 3. Procedure PARTITION.

We can partition the preimage class D with respect to any $r-1$ classes of $C_1, \ldots, C_r$, and the result will be the same because $D_s = D-(D_1+\cdots+D_{s-1}+D_{s+1}+\cdots+D_r)$. Since we can choose any $r-1$ classes, we partition with respect to $r-1$ small ones except the largest at stage $l+1$.

Procedure PARTITION in Figure 3 shows the partitioning algorithm. Since there are $O(n)$ classes, we represent each class by a number k for $1 \leq k \leq cn$. Each class is implemented by a doubly linked list of its elements in ascending order. When we partition with respect to C in Figure 3, the classes D which are partitioned with respect to C can be easily identified because D is a class that contains i such that $i+1$ is in C. See [8] for more details of implementation, where the time complexity of PARTITION is shown to be proportional to the sum of the sizes of classes C with respect to which the partitioning is made. Initially ($l = 1$), all classes of E_1 are in SMALL; i.e., all positions in x belong to SMALL. Consider a position i in a class D of E_l. Suppose that D is split at stage l and the subclass D' containing i is put in SMALL. Then $|D'| \leq |D|/2$. Therefore, one position cannot belong to SMALL more than $\log n$ times. Since there are n positions, Procedure PARTITION takes $O(n \log n)$ time.

This partitioning is similar to the single function partitioning due to Hopcroft [10], [1]. The former can be viewed as a special case of the latter in which $f(i) = i+1$ with the following two exceptions:

1. One position is excluded at each stage.
2. Each stage must be separated from another, because each stage deals with equivalence relation E_l. (Thus the algorithm in [1] cannot be applied directly to our partitioning.)

Crochemore [8] used this partitioning for computing all repetitions in a string. Although the approach of Apostolico and Preparata [4] computes the same equivalence classes with their elements sorted, the partitioning method in Figure 3 is simpler and more elegant ([4] uses quite complicated data structures).

We compute equi-sets from the start-sets as follows. When a new class A (which is a start-set) is formed at stage l, we record the number $l+1$ with the class A (i.e., A is a class of E_{l+1}). The class A will be either split later or remain unchanged until PARTITION stops (if A is a singleton set or stage p is reached). If the class A is split into smaller classes at stage l' for $l' > l$, the equi-set associated with A is the set of strings of length $l+1$ to l' whose start-set is A. If A remains unchanged, let e_t be the largest element in A. The equi-set associated with A is the set of strings of length $l+1$ to $\min(p-1, n-e_t+1)$ whose start-set is A.

We now obtain candidate-sets from the equi-sets. Let $S = \{w_l, w_{l+1}, \ldots, w_g\}$ be an equi-set such that $|w_i| = i$ for $l \le i \le g$, and let its associated start-set be A. From the start-set A, the differences of consecutive positions can be computed. For example, if $A = \{3, 5, 9, 12\}$, the differences are $2, 4, 3$. During the partitioning, we maintain only the maximum difference between elements for each equivalent class that has at least two elements. Since each class is implemented by a doubly linked list of increasing elements, and elements are only deleted once a class is formed (there are no insertions), computing maximum differences does not take more time than the partitioning. Let d be the maximum difference in A; $d = 0$ if A is a singleton set. Let $h = \max(d, l)$, and let e_1 and e_t be the first and last start positions of w_h in the given string x, respectively. If $h \le g$, then w_i for $h \le i \le g$ a-covers the substring x' of x, where $x' = x_{e_1} \cdots x_{e_t+i-1}$.

To complete the computation of a candidate-set with respect to v, we eliminate each w_i such that the length of the head or tail with respect to w_i is $\ge |w_i| = i$. The length of the head (resp. tail) of x with respect to w_i is $e_1 - 1$ (resp. $n - e_t + 1 - i$). Hence a candidate w_i must satisfy $i \ge e_1$ and $i > n - e_t + 1 - i$. Let $f = \max(h, e_1, \lceil (n - e_t + 2)/2 \rceil)$. If $f > g$, then no strings of S are candidates. Otherwise, $\{w_f, \ldots, w_g\}$ is a candidate-set. This computation takes constant time for each equi-set S; $O(n)$ time for all equi-sets.

4. Finding Hard Seeds. To compute type-A seeds from candidate-sets below, we need the following processing on the given string x. For an example with $x = ababaabb$, see Figure 4. For each position i of x, let $F[i]$ be the length of the maximal proper prefix of $x_1 \cdots x_i$ which is also a suffix of $x_1 \cdots x_i$, and let $B[i]$ be the length of the maximal proper suffix of $x_i \cdots x_n$ which is a prefix of $x_i \cdots x_n$.

For $1 \le i \le n$, let $F^+[i] = i - F[i]$. Then $F^+[i] > 0$ for all i. Consider the values of F from left to right. Let $F[i] = r$. Then $F[i+1] = r+1$ if $x_{i+1} = x_{r+1}$, and $F[i+1] \le r$ otherwise. That is, the F values increase by 1 or are nonincreasing. Therefore, the F^+ values are nondecreasing from left to right. For $1 \le i \le n$, let $MF^+[i]$ be the largest j such that $F^+[j] \le i$. Similarly, let $B^+[i] = (n - i + 1) - B[i]$. Then $B^+[i] > 0$ for all i, and the B^+ values are nondecreasing from right to left. For $1 \le i \le n$, let $MB^+[i]$ be the smallest j such that $B^+[j] \le i$.

	a	b	a	b	a	a	b	b
Index :	1	2	3	4	5	6	7	8
F :	0	0	1	2	3	1	2	0
F^+ :	1	2	2	2	2	5	5	8
MF^+ :	1	5	5	5	7	7	7	8

Fig. 4. An example of F, F^+, and MF^+.

The arrays $F[1..n]$ and $B[1..n]$ can be computed in $O(n)$ time by the preprocessing of the Knuth–Morris–Pratt algorithm [11]. The array MF^+ (resp. MB^+) is computed in $O(n)$ time by scanning the F^+ values from left to right (resp. the B^+ values from right to left).

We now describe how to find type-A seeds from each candidate-set $S = \{w_i \mid f \leq i \leq g\}$ in constant time. Since there are $O(n)$ candidate-sets, it will take $O(n)$ time for all candidate-sets. Let e_i, $1 \leq i \leq t$, be the start position of the ith occurrence of the candidates in S. By the definition of candidates, the head and the tail of x with respect to w_f are shorter than w; i.e., $e_1 \leq f$ and $n - e_t + 1 < 2f$.

We first check the tails with respect to the candidates of S. Let $k = (n - e_t + 1) - B[e_t]$.

(i) If $k > g$, then no candidate w_i, $f \leq i \leq g$, covers the tail of x (with respect to w_i). Therefore, none of the candidates are seeds.
(ii) If $k < f$, then every candidate w_i in S covers the tail of x.
(iii) If $f \leq k \leq g$, then each candidate w_i, $k \leq i \leq g$, covers the tail of x.

In cases (ii) and (iii), let $h = \max(k, f)$ (i.e., w_h is the shortest candidate which covers the tail of x).

To compute type-A seeds among the candidates $w_h, \ldots, w_g$, it remains to find those candidates w_i whose l-size is greater than or equal to its r-size. (Since r-size is ≥ 0, it will imply that w_i covers the head.) For $h \leq i \leq g$, the r-size of w_i is $i - k$. Note that $e_1 + i - 1$ is the end position of the first occurrence of w_i in x for $h \leq i \leq g$. For $h \leq i \leq g$, the l-size of w_i is $F[e_1 + i - 1] - (e_1 - 1)$. We need to find candidates whose l-size is greater than or equal to its r-size; i.e., w_i for $h \leq i \leq g$ such that $F[e_1 + i - 1] - e_1 + 1 \geq i - k$ (or $(e_1 + i - 1) - F[e_1 + i - 1] = F^+[e_1 + i - 1] \leq k$). Let s be the integer such that $e_1 + s - 1 = MF^+[k]$; i.e., s is the largest integer such that the l-size of w_s is larger than or equal to its r-size.

(i) Case $s < h$: The l-size of w_i, $h \leq i \leq g$, is less than its r-size. Therefore, there are no type-A seeds in S.
(ii) Case $s \geq h$: The l-size of w_i, $h \leq i \leq \min(s, g)$, is greater than or equal to its r-size. Therefore, $w_h, \ldots, w_{\min(s,g)}$ are type-A seeds.

Since there are $O(n)$ candidate-sets and the type-A seeds from a candidate-set are reported in constant space, the output size of the algorithm is $O(n)$.

For type-B seeds, we compute the sets of end positions of substrings of x. Checking the heads and tails is symmetric to the case of type-A seeds. From the discussion so far, we have the following theorem.

THEOREM 2. *All the hard seeds of the given string x can be found in $O(n \log n)$ time.*

5. Conclusion. We have presented an $O(n \log n)$-time algorithm for finding all the seeds of a given string of length n. Recently, Ben-Amram *et al.* [5] gave a parallel algorithm for the same problem that runs in $O(\log n)$ time using n processors on a CRCW PRAM.

The existence of a linear-time algorithm for the general string covering problem is an open problem. Another interesting problem is to find segments which approximately cover a given string, together with the problem of computing the approximate period of a given string. Here we considered only the exact version of this problem, where all segments must be the same.

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, *Theoret. Comput. Sci.* **119** (1993), 247–265.

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

[4] A. Apostolico and F. P. Preparata, Optimal off-line detection of repetitions in a string, *Theoret. Comput. Sci.* **22** (1983), 297–315.

[5] A. M. Ben-Amram, O. Berkman, C. S. Iliopoulos, and K. Park, The subtree max gap problem with application to parallel string covering, *Proc. 5th ACM–SIAM Symp. on Discrete Algorithms*, 1994, pp. 501–510.

[6] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas, The smallest automaton recognizing the subwords of a text, *Theoret. Comput. Sci.* **40** (1985), 31–55.

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

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

[9] M. Crochemore, Transducers and repetitions, *Theoret. Comput. Sci.* **45** (1986), 63–86.

[10] J. E. Hopcroft, An $n \log n$ algorithm for minimizing states in a finite automaton, in Kohavi and Paz, eds., *Theory of Machines and Computations*, Academic Press, New York, 1971, pp. 189–196.

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