On-line Algorithms for k-Covering

Costas S. Iliopoulos[1*] and W.F. Smyth[2**]

[1] Dept. Computer Science, King's College London, London WC2R 2LS, England, and School of Computing, Curtin University of Technology, Perth 6845, WA, Australia
csi@dcs.kcl.ac.uk,
www.dcs.kcl.ac.uk/staff/csi

[2] Algorithms Research Group, Dept. of Computing & Software, McMaster University, Hamilton, Ontario, Canada L8S 4L7
and School of Computing, Curtin University, Perth WA 6845, Australia
smyth@mcmaster.ca
www.dcss.mcmaster.ca/~ bill/cv.shtml

Abstract. An $O(n^2(n-k))$ on-line algorithm for computing a minimum set of k-covers for a given string of length n is presented. A straightforward modification of the algorithms yields $O(kn^2(n-k))$ algorithms for computing a minimum set of k-covers and k-segments for a given circular string of length n. We show further that the number of such minimum sets of k-covers may be exponential. Similar time complexity bounds hold for computing the minimum sets of k-segments and k-seeds.

Keywords: String algorithms, string matching, dynamic programming, molecular sequences, data compression.

1 Introduction

A substring w of x is called a *period* of x if x is a prefix of a string that can be constructed by concatenating two or more copies of w. A substring w of x is called a *cover* of x if x can be constructed by concatenations and superpositions of w. A substring w of x is called a *seed* of x if there exists a superstring of x which can be constructed by concatenations and superpositions of w. A seed can thus be thought of as either a generalized period or a generalized cover. For example, abc is a period of $abcabcabca$, $abca$ is a cover of $abcabcaabca$, and $abca$ is a seed of $abcabcaabc$.

There has been much recent study of problems relating to covers and seeds. Apostolico, Farach and Iliopoulos [AFI91] gave a linear-time algorithm for computing the shortest cover of a given string. Breslauer [Br92] presented a linear-time on-line algorithm for the same problem. Moore and Smyth [MS94] presented

* Partially supported by the EPSRC grant GR/J 17844.

** 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.

a linear-time algorithm to compute all the covers of a given string, and then Li and Smyth [LS99] discovered a linear-time algorithm to compute all the covers of every prefix of a string. In parallel computation, Breslauer [Br94] gave PRAM algorithms for the shortest-cover problem. Iliopoulos and Park [IP94] gave an optimal $O(\log\log n)$-time (thus work-time optimal) algorithm for the shortest-cover and all-covers problems. Iliopoulos, Moore and Park [IMP93] introduced the notion of seeds and gave an $O(n \log n)$-time algorithm for computing all the seeds of a given string of length n. For the same problem Ben-Amram, Berkman, Iliopoulos and Park [BBIP94] presented a parallel algorithm that requires $O(\log n)$ time and $O(n \log n)$ work. Apostolico and Ehrenfeucht [AE93] considered yet another problem related to covers.

In the hybridization approach [DC92] to analysis of a DNA sequence (or string) x of $n = 300$-500 base pairs, the base pairs are determined by comparing sections (or substrings) of x of some fixed length k (typically in the range 5-14) with a "chip" U_k which normally contains all possible strings of length k. Thus, in general, for an alphabet Σ of cardinality $\sigma = |\Sigma|$, the number of strings in U_k is σ^k; in particular, for a DNA sequence, $\Sigma = \{C, G, A, T\}$ and $\sigma = 4$. Clearly each such chip, or set of substrings, guarantees that every position in x is contained in a substring which matches one of the substrings on the chip: extending the above definition, we say then that the set U_k *covers* (or is a *cover* of) x. It is clear that in order to guarantee that every string x can be covered, the chip U_k must contain *all* strings of length k; otherwise, $y \notin U_k$ would imply that no string $x = yx'$ or $x'y$ could be covered. Related problems on hybridization chips are discussed in [DS96], [PL93] and [SS93].

Here we consider the problem of computing a *minimum set of k-covers* of a string. That is, given a string x and an integer $k < |x|$, compute a set $C = \{w_1, w_2, \ldots, w_m\}$ of substrings of x such that:

(i) every w_i is of length k;
(ii) the set C covers the string x;
(iii) the number $m = |C|$ of such substrings is the smallest possible.

Here we present an $O(n^2(n-k))$ time on-line algorithm for computing a minimum set of k-covers for all prefixes of a given string x of length n. This gives rise to an $O(kn^2(n-k))$ algorithm for computing the cardinality of a minimum set of k-covers for a given circular string of length n. Easy extensions to these algorithms allow us to compute at least one such minimum set of k-covers. However, we do not compute all such minimum sets, since it turns out that the cardinality of such sets may be as much as $\Theta(c^n)$ for some constant $c > 1$. We also outline how these algorithms can modified in order to provide solutions to the problems of computing a minimum set of k-segments and k-seeds of a given string (see Section 6).

2 Preliminaries

A *string* is a sequence of zero or more symbols drawn from an alphabet Σ. The set of all strings over Σ is denoted by Σ^*. The string of length zero is the *empty* *string* ϵ; a string x of length $n > 0$ is represented by $x_1 x_2 \cdots x_n$, where $x_i \in \Sigma$ for $1 \leq i \leq n$. A string w is a *substring* of x if $x = uwv$ for $u, v \in \Sigma^*$. More precisely, let $i \leq n$ and $j \leq n$ denote nonnegative integers: if $1 \leq i \leq j$, $x[i..j]$ denotes the substring of x that starts in position i and has length $j - i + 1$; otherwise, $x[i..j] = \epsilon$. 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 concatenation 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 $i \geq 1$ (that is, such that x has a suffix equal to a prefix of y), the string $x_1 \cdots x_n y_{i+1} \cdots y_m$ is said to be a *superposition* of x and y. Alternatively, we may say that x *overlaps* with y.

Next we consider some basic facts about minimum sets of k-covers. A first observation is the following:

Fact 1
The strings $x[1..k]$ and $x[n-k+1..n]$ of x are both elements of every minimum set of k-covers of the string x. □

One can see that the strings $C_i = x[ik+1..ik+k]$, $i = 0, 1, \ldots, \lfloor n/k \rfloor - 1$, together with the string $x[n-k+1..n]$ cover $x = x_1 \cdots x_n$, leading to the following fact:

Fact 2
The cardinality of a minimum set of k-covers of a string of length n is at most $\lceil n/k \rceil$. □

Consider the string $x = abcdefg$. The sets

$$\{abc, bcd, efg\}, \{abc, cde, efg\}, \{abc, def, efg\}$$

are all minimum sets of 3-covers of x. Thus we have:

Fact 3
A minimum set of k-covers for a string x is not unique. □

The reason for not computing the minimum sets themselves is that there may be an exponential number of them, as the next lemma shows.

Lemma 1. *Let x denote a string of length n whose symbols are all distinct; that is, for every pair of positions i and i' in x, $x[i] = x[i']$ iff $i = i'$. Let $n = hk - j$ for integers $h \geq 2$ and $j \in 0..k-1$, and let $N_{h,j}$ denote the number of distinct minimum sets of k-covers of x. Then*

(a) $N_{h,j} = \sum_{0 \leq i \leq j} N_{h-1,i}$ *for every* $h \geq 3$*;*
(b) $N_{h,j} \in \Theta((j+1)^{h-1})$.

Proof. Observe that $N_{2,j} = 1$ for every value of j, and that $N_{3,j} = j+1$. Thus (a) holds for $h = 3$. More generally, observe that in a minimum set there are exactly $j+1$ possible distinct choices for the k-strings that concatenate/overlap with $x[1..k]$. These choices are

$$x[k+1..2k], x[k..2k+1], \ldots, x[k-j+1..2k-j],$$

leaving, respectively, suffixes of length

$$(h-2)k-j, (h-2)k-j+1, \ldots, (h-2)k$$

to be covered. The number of distinct minimum set of k-covers that cover these suffixes together with the prefix $x[1..k]$ is, respectively,

$$N_{h-1,j}, N_{h-1,j-1}, \ldots, N_{h-1,0}.$$

Thus (a) is established.

Observe finally that since (a) holds for all h and j, it follows that (b) also holds. □

Since $h = \lceil n/k \rceil$, it follows from Lemma 1(b) that the number of distinct minimum sets of k-covers may be exponential in n for $j > 0$. This is a major complicating factor in the design of an iterative algorithm to compute a minimum set of k-covers for x, since we will want to avoid the necessity of inspecting a possibly exponential number of sets of k-covers for prefixes of x.

3 The Main Ideas

Initially the algorithm finds the trivial cardinalities (equal to 1 or 2) of minimum sets of k-covers for the prefixes $x[1..k+1], x[1..k+2], \ldots, x[1..2k]$ of x. The algorithm works iteratively, first computing the cardinality of a minimum set of k-covers for the prefix $x[1..i]$ of the given string x for some integer i; it then proceeds to compute the same function for $x[1..i+1]$. The algorithm is based upon the following two main ideas:

1. Fact 1 stating that the minimum set of k-covers contains the suffix of length k. This is used as a yardstick to find a minimum set.
2. For $i \geq k$, every minimum set of k-covers for $x[1..i+1]$ depends only on the minimum sets computed for the previous k positions; that is, the minimum sets of k-covers for $x[1..i], x[1..i-1], \ldots, x[1..i-k+1]$.

Lemma 2. *For $i \geq 2k$ and $l = 1, 2, \ldots$, let $C_{i,l}$ denote the distinct minimum sets of k-covers for $x[1..i]$ and let c_i denote their cardinality. Also let $s = x[i-k+2..i+1]$. Then*

$$c_{i+1} = \min_{\substack{i-k+1 \leq j \leq i \\ \textit{every } l}} |C_{j,l} \cup \{s\}|. \qquad (3.1)$$

Proof. It is clear that every set $C_{j,l} \cup \{s\}$ is a k-cover for $x[1..i+1]$. Therefore

$$c_{i+1} \leq \min_{\substack{i-k+1 \leq j \leq i \\ \text{every } l}} |C_{j,l} \cup \{s\}|.$$

Assume then that

$$c_{i+1} < \min_{\substack{i-k+1 \leq j \leq i \\ \text{every } l}} |C_{j,l} \cup \{s\}| \tag{3.2}$$

and let C' denote a minimum set of k-covers for $x[1..i+1]$. From Fact 1, we know that s is a member of C' . We now consider the way that C' covers $x[1..i+1]$. The string s occurs at position $i-k+2$ of $x[1..i+1]$. Since C' covers $x[1..i+1]$, there exists at least one string $u \in C'$ that occurs at position r, for some $i-2k+2 \leq r \leq i-k+1$. Thus C' is a (not necessarily minimum) set of k-covers for $x[1..r+k-1]$, so that

$$|C'| = c_{i+1} \geq c_{r+k-1}.$$

Now if $\hat{C}$ is a minimum set of k-covers for $x[1..r+k-1]$ that has s as an element, then

$$|C'| = c_{i+1} \geq c_{r+k-1} = |\hat{C} \cup \{s\}| \geq \min_{\substack{i-k+1 \leq j \leq i \\ \text{every } l}} |C_{j,l} \cup \{s\}|$$

in contradiction to (3.2). If there is no minimum set of k-covers for $x[1..r+k-1]$ that has s as an element, then C' cannot be a minimum set of k-covers for $x[1..r+k-1]$. Hence

$$|C'| = c_{i+1} > c_{r+k-1}$$

or equivalently

$$|C'| = c_{i+1} \geq c_{r+k-1} + 1 \geq \min_{\substack{i-k+1 \leq j \leq i \\ \text{every } l}} |C_{j,l} \cup \{s\}|,$$

which also contradicts (3.2). □

The next result is an immediate consequence of Lemma 2:

Lemma 3. *For $i \geq 2k$, every minimum set $C_{i+1,l}$ is a superset of some minimum set $C_{j,l'}$, $i-k+1 \leq j \leq i$.*

Proof. A result of the fact that there exists $j \in i-k+1..i$ and some integer l' such that $C_{i+1,l} = C_{j,l'} \cup \{s\}$, where $s = x[i-k+2..i+1]$. □

Finally, we show that c_{i+1} takes values only in a very limited range:

Lemma 4. *For $i \geq 2k$, suppose that $C_{i+1,l} \supseteq C_{j,l'}$ for some $j \in i-k+1..i$ and some l'. As before, let $s = x[i-k+2..i+1]$. Then $c_{i+1} = c_j$ if and only if $s \in C_{j,l'}$; otherwise, $c_{i+1} = c_j + 1$.*

Proof. Also an immediate consequence of Lemma 3.1. □

4 An Outline of the Algorithm

From what we have seen so far, it is clear that an algorithm to compute a minimum set of k-covers (indeed, *all* minimum sets of k-covers) could easily be implemented by computing, for each position $i+1$ in x, the minimum sets

$$C_{i+1,l} = C_{j,l'} \cup \{x[i+k-1..i]\}$$

for appropriate choices of $j \in i-k+1..i$. Unfortunately, such an algorithm would be inefficient since, as we have seen in Lemma 1, the number of such minimum sets for any value of j may be exponential in j.

To achieve an efficient algorithm, we store for each position i of x an array that identifies all the k-strings that occur in *at least one* of the minimum sets $C_{i,l}$. Thus the storage associated with each position is $O(n)$ rather than $O(i^n)$. More precisely, the storage required by the algorithm is as follows:

1. for every $i \in k+1..n$, the value c_i, the cardinality of every minimum set $C_{i,l}$ of k-covers of $x[1..i]$;
2. for every $i \in k+1..n$, a boolean array $A_i = A_i[k..i]$, where $A_i[j] = \texttt{TRUE}$ if and only if the k-string $x[j-k+1..j]$ is an element of at least one of the minimum sets $C_{i,l}$;
3. a global array $L = L[k..n]$ that implements circular lists among equal k-strings in x — that is, if $L[i] = j$ then $x[i-k+1..i] = x[j-k+1..j]$ and the sequence $i, L[i], L[L[i]], \ldots, L^m[i] = i$ identifies all m k-strings in x that are equal to $x[i-k+1..i]$. Note that it may occur that $m \in O(n)$.

The array L can be computed in $O(n \log n)$ time using Crochemore's algorithm [C81] or suffix trees ([A85],[W73]). For $i \in k+1..2k$, the other data structures are

$$\begin{aligned}
c_i &\leftarrow 1,\ \text{if } x[i-k+1..i] = x[1..k];\\
c_i &\leftarrow 2,\ \text{otherwise};\\
A_i[j] &\leftarrow \texttt{TRUE},\ \text{if } j = i \text{ or } k;\\
A_i[j] &\leftarrow \texttt{FALSE},\ \text{otherwise.}
\end{aligned}$$

Note that storage in each array A_i differentiates between different occurrences of identical k-strings; thus, for example, even though $A_i[i]$ is always $\texttt{TRUE}$, there may nevertheless exist $j < i$ such that $x[k-j+1..j] = x[k-i+1..i]$ and $A_i[j] = \texttt{FALSE}$.

After initialization, the algorithm iteratively computes c_{i+1} and A_{i+1} for every $i+1 = 2k+1, 2k+2, \ldots, n$. The computation of c_{i+1} is carried out by first using L to mark every position $j \in i-k+1..i$ such that $A_j[q] = \texttt{TRUE}$ for some q such that $x[q-k+1..q] = x[i-k+2..i+1]$; if there exists such a q, then $\texttt{MARK}[j]$ is set $\texttt{TRUE}$, otherwise $\texttt{FALSE}$. This calculation requires $\Omega(k)$ and $O(nk)$ time. Then in the second stage, the algorithm computes

$$c_{i+1} \leftarrow \min_{i-k+1 \le j \le i} \{c_j \text{ if } \texttt{MARK}[j] = \texttt{TRUE}, \quad c_j + 1 \text{ otherwise}\}. \tag{4.1}$$

By Lemmas 2, 3 and 4 this second calculation is correct; since it requires $O(k)$ time, the computation of c_{i+1} can be completed in $O(nk)$ time.

We turn now to the computation of A_{i+1} which we suppose to be initialized to `TRUE` in positions k and $i+1$; otherwise, to `FALSE`. As above, letting $s = x[i-k+2..i+1]$, we find that two cases may arise for *each* value $j \in i-k+1..i$ that contributes to the minimum in (4.1):

- for every occurrence $x[q-k+1..q]$ of s, $A_j[q] = \texttt{FALSE}$ (this of course includes the case in which there is no such q);
- for at least one occurrence $x[q-k+1..q]$ of s, $A_j[q] = \texttt{TRUE}$.

Observe that, using the array L, $O(n)$ time is required to distinguish these cases. In the first case, the computation of A_{i+1} is straightforward, since by Lemma 4 all k-strings referenced in A_j must contribute to the minimum:

$$\text{for every } r \text{ such that } A_j[r] = \texttt{TRUE}, \quad A_{i+1}[r] \leftarrow A_j[r].$$

This computation requires $O(n)$ time for each value of j that contributes to (4.1), thus a total of $O(nk)$ time.

In the second case the computation of A_{i+1} is more difficult: first, every occurrence of s in any set of k-covers of $x[1..j]$ should be included in a minimum set of k-covers for $x[1..i+1]$; second, exactly those k-strings that occur together with s in a minimum set of k-covers for $x[1..j]$ should also be included. In order to identify these k-strings, we consider all pairs of integers q and r defined as follows:

$$\begin{aligned} A_j[q] = A_j[r] &= \texttt{TRUE}; \\ x[q-k+1..q] &= s; \\ x[r-k+1..r] &\neq s. \end{aligned}$$

For $q < r$, we let $A = A_r[q]$; otherwise, we let $A = A_q[r]$. If $A = \texttt{TRUE}$, then we know that $x[q-k+1..q]$ and $x[r-k+1..r]$ occur together, and accordingly we set

$$A_{i+1}[r] \leftarrow \texttt{TRUE}; \quad A_{i+1}[q] \leftarrow \texttt{TRUE}.$$

Since there may be $O(n)$ values of q and $O(n)$ values of r to be considered, this second case in the computation of A_{i+1} requires $O(n^2)$ time.

Putting together the components of the algorithm described above, we see that the total time including initialization and all values $i+1 \in 2k+1..n$ will be

$$O(n \log n + k^2 + (n-2k)(nk + n + nk + n^2)).$$

Hence:

Theorem 1. *The above algorithm correctly computes the cardinality of a minimum set of k-covers for each of the prefixes of given string x of length n in $O(n^2(n-k))$ time.* □

Finally, observe that in order to determine a minimum set of k-covers for any position i, we need only determine some integer $j \in i-k..i-1$ such that $A_i[j] =$ `TRUE`, a calculation requiring $O(k)$ time. Then we just repeat the calculation recursively for j. Thus

Theorem 2. *A minimum set of k-covers for each of the prefixes of a given string x of length n can be computed in $O(n^2(n-k))$ time.* □

5 Cyclic Strings

Here we consider the problem of computing a minimum set of k-covers for a circular string.

One can use the on-line algorithm of the previous section to derive a cubic-time algorithm for computing the cardinality of the minimum set of k-covers for a circular string as follows:

1. Let $x = x_1 x_2 ... x_n$. We define

$$R_{i,j}(x) = x_i x_{i+1} ... x_n x_1 ... x_{i-1} x_i x_{i+1} ... x_{i+j}$$

2. Compute the cardinality of the minimum set of k-covers of all $R_{i,j}(x), \{i,j\} \in \{1, ...k\}$. Note that $R_{i,j}$ is a prefix of $R_{i,k}$, for $j = 1, ..., k$, and thus an application (for each i) of the algorithm above suffices.
3. The minimum of the cardinalities of the minimum sets of k-covers of all the above prefixes is the required cardinality of a minimum set of k-covers for the cyclic string x.

Theorem 3. *The computation of the cardinality of a minimum set of k-covers for a circular string x of length n requires $O(kn^2(n-k))$ time.* □

6 Computing a minimum set of k-segments and k-seeds

A variant of the k-covers problem is the problem of computing a *minimum set of k-segments* of a string. That is, given a string x and an integer $k < |x|$, compute a set $C = \{w_1, w_2, \ldots, w_m\}$ of substrings of x such that:

(i) every w_i is of length less or equal to k;
(ii) the set C covers the string x;
(iii) the number $m = |C|$ of such substrings is the smallest possible.

The algorithm for k-covering is based upon Lemma 2 and in particular upon evaluating formula (3.1). We can show that a similar lemma holds for the minimum sets of k-segments:

Lemma 5. *For $i \geq 2k$ and $l = 1, 2, \ldots$, let $C_{i,l}$ denote the distinct minimum sets of k-segments for $x[1..i]$ and let c_i denote their cardinality. Also let $s = x[i-k+2..i+1]$. Then*

$$c_{i+1} = \min_{\substack{i-k+1 \leq j \leq i \\ i-j \leq m \leq k \\ \textit{every } l}} |C_{j,l} \cup \{\text{suffix}_m(s)\}|. \tag{6.1}$$

Our algorithm for computing the cardinality of the minimum set of k-covers can be easily modified to compute a mimimum set of k-segments using (6.1). The time required is $O(kn^2(n-k))$.

Similar results can be obtained for the problem of computing a *minimum set of k-seeds* of a string. That is, given a string x and an integer $k < |x|$, compute a set $C = \{w_1, w_2, \ldots, w_m\}$ of substrings of x such that:

(i) every w_i is of length less or equal to k;
(ii) the set C is set of seeds for the string x (i.e. there exists a superstring of x constructed with concatenations and superpositions of strings in C.)
(iii) the number $m = |C|$ of such substrings is the smallest possible.

7 Conclusions

Here we have presented essentially cubic-time algorithms for computing the cardinality of a minimum set of k-covers. It would be interesting to determine whether or not the running times are tight. An implementation in C of the basic k-covers algorithm, written by Lu Yang, can be found at

`www.dcss.mcmaster.ca/~bill/kcovers.c`

References

[A85] A. Apostolico, The myriad virtues of subword trees, in *Combinatorial Algorithms on Words*, Nato ASI series, Computer Systems and Sciences, Vol 12, Springer verlag, Berlin, (1985) 85–96.

[AE93] A. Apostolico and A. Ehrenfeucht, Efficient detection of quasiperiodicities in strings, *Theoret. Comput. Sci.* 119 (1993) 247–265.

[AFI91] A. Apostolico, M. Farach and C. S. Iliopoulos, Optimal superprimitivity testing for strings, *Inform. Process. Lett.* 39 (1991) 17–20.

[BBIP94] 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. Discrete Algorithms* (1994) 501–510.

[Br92] D. Breslauer, An on-line string superprimitivity test, *Inform. Process. Lett.* 44 (1992) 345–347.

[Br94] D. Breslauer, Testing string superprimitivity in parallel, *Inform. Process. Lett.* 49-5 (1994) 235–241.

[C81] Maxime Crochemore, An optimal algorithm for computing all the repetitions in a word, *Inform. Process. Lett.* 12-5 (1981) 244–248.

[DC92] R. Drmanac & R. Crkvenjakov, Sequencing by hybridization (SBH) with oligonucleotide probes as an integral approach for the analysis of complete genomes, *Int. J. of Genome Research 1-1* (1992) 59-79.

[DS96] Art M. Duval and W. F. Smyth, Covering a circular string with substrings of fixed length, *Int. Journal of Foundations of Computer Science* 7-1 (1996) 87–93.

[IMP93] C. S. Iliopoulos, D. W. G. Moore and K. Park, Covering a string, *Proc. 4th Symp. Combinatorial Pattern Matching*, Lecture Notes in Computer Science 684 (1993) 54–62.

[IP94] C. S. Iliopoulos and K. Park, An optimal $O(\log\log n)$-time algorithm for parallel superprimitivity testing, *J. Korea Information Science Society* 21-8 (1994) 1400–1404.

[LS94] Yin Li and W. F. Smyth, Computing the cover array in linear time, *Algorithmica*, to appear.

[MS94] D. W. G. Moore and W. F. Smyth, Computing the covers of a string in linear time, *Inform. Process. Lett.* 50-5 (1994) 239-246.

[PL93] Pavel A. Pevzner & Robert J. Lipshutz, Toward DNA sequencing by hybridization, preprint (1993).

[SS93] Steven S. Skiena & Gopalakrishnan Sundaram, Reconstructing strings from substrings, preprint (1993).

[W73] P. Weiner, Linear pattern matching algorithm, *Proc 14-th IEE Sympos. Switching and Automata Theory* (1973) pp1-11.