Computing Covers Using Prefix Tables *

Ali Alatabbi[1], M. Sohel Rahman **[2], and W. F. Smyth[1,3,4]

[1] Department of Informatics, King's College London
ali.alatabbi@kcl.ac.uk
[2] Department of Computer Science & Engineering
Bangladesh University of Engineering & Science
msrahman@cse.buet.ac.bd
[3] Algorithms Research Group, Department of Computing & Software
McMaster University
smyth@mcmaster.ca
[4] School of Engineering & Information Technology
Murdoch University, Western Australia

Abstract. An ***indeterminate string*** $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on an alphabet Σ is a sequence of nonempty subsets of Σ; $\boldsymbol{x}$ is said to be ***regular*** if every subset is of size one. A proper substring $\boldsymbol{u}$ of regular $\boldsymbol{x}$ is said to be a ***cover*** of $\boldsymbol{x}$ iff for every $i \in 1..n$, an occurrence of $\boldsymbol{u}$ in $\boldsymbol{x}$ includes $\boldsymbol{x}[i]$. The ***cover array*** $\boldsymbol{\gamma} = \boldsymbol{\gamma}[1..n]$ of $\boldsymbol{x}$ is an integer array such that $\boldsymbol{\gamma}[i]$ is the longest cover of $\boldsymbol{x}[1..i]$. Fifteen years ago a complex, though nevertheless linear-time, algorithm was proposed to compute the cover array of regular $\boldsymbol{x}$ based on prior computation of the border array of $\boldsymbol{x}$. In this paper we first describe a linear-time algorithm to compute the cover array of regular $\boldsymbol{x}$ based on the prefix table of $\boldsymbol{x}$. We then extend this result to indeterminate strings.

1 Introduction

The idea of a ***quasiperiod*** or ***cover*** of a string $\boldsymbol{x}$ was introduced almost a quarter-century ago by Apostolico & Ehrenfeucht [4]: a proper substring $\boldsymbol{u}$ of $\boldsymbol{x}$ such that every position in $\boldsymbol{x}$ lies within an occurrence of $\boldsymbol{u}$. Thus, for example, $\boldsymbol{u} = aba$ is a cover of $\boldsymbol{x} = ababaababa$. In [5] a linear-time algorithm was described to compute the shortest cover of $\boldsymbol{x}$; this contribution was followed by linear-time algorithms to compute

- the shortest cover of every prefix of $\boldsymbol{x}$ [9];
- all the covers of $\boldsymbol{x}$ [17, 18];
- all the covers of every prefix of $\boldsymbol{x}$ [16].

* This work was supported in part by the Natural Sciences & Engineering Research Council of Canada.

** Partially supported by a Commonwealth Academic Fellowship and an ACU Titular Fellowship, both funded by the UK Government. Currently on a sabbatical leave from BUET.

A ***border*** of a string $\boldsymbol{x}$ is a possibly empty proper prefix of $\boldsymbol{x}$ that is also a suffix of $\boldsymbol{x}$. (Thus a cover of $\boldsymbol{x}$ is necessarily also a border of $\boldsymbol{x}$.) In the ***border array*** $\boldsymbol{\beta} = \boldsymbol{\beta}[1..n]$ of the string $\boldsymbol{x} = \boldsymbol{x}[1..n]$, $\boldsymbol{\beta}[i]$ is the length of the longest border of $\boldsymbol{x}[1..i]$. Since for $\boldsymbol{\beta}[i] \neq 0$, $\boldsymbol{\beta}[\boldsymbol{\beta}[i]]$ is the length of a border of $\boldsymbol{x}$ as well as the length of the longest border of $\boldsymbol{x}[1..\boldsymbol{\beta}[i]]$ [2, 20], it follows that $\boldsymbol{\beta}$ provides all the borders of every prefix of $\boldsymbol{x}$. For example:

$$\begin{array}{rl} & \scriptstyle 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10 \\ \boldsymbol{x} = & a\ b\ a\ b\ a\ b\ a\ a\ b\ a \\ \boldsymbol{\beta} = & 0\ 0\ 1\ 2\ 3\ 4\ 5\ 1\ 2\ 3 \end{array} \tag{1}$$

As shown in [16], the ***cover array*** $\boldsymbol{\gamma}$ has a similar cascading property, giving the lengths of all the covers of every prefix of $\boldsymbol{x}$ in a compact form:

$$\begin{array}{rl} & \scriptstyle 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10 \\ \boldsymbol{\gamma} = & 0\ 0\ 0\ 2\ 3\ 4\ 5\ 0\ 0\ 3 \end{array}$$

Here $\boldsymbol{x}[1..7]$ has covers $\boldsymbol{u_1} = \boldsymbol{x}[1..5] = ababa$ and $\boldsymbol{u_2} = \boldsymbol{x}[1..3] = aba$, while the entire string $\boldsymbol{x}$ has cover $\boldsymbol{u_2}$. The main result of [16] is an algorithm that computes $\boldsymbol{\gamma} = \boldsymbol{\gamma}[1..n]$ from $\boldsymbol{\beta} = \boldsymbol{\beta}[1..n]$ in $\Theta(n)$ time, while making no reference to the underlying string $\boldsymbol{x}$.

The results outlined above all apply to a ***regular string*** — that is, a string $\boldsymbol{x}$ such that each entry $\boldsymbol{x}[i]$ is constrained to be a one-element subset of a given set Σ called the ***alphabet***. In this paper we show how to extend these ideas and algorithms to an ***indeterminate string*** $\boldsymbol{x}$ — that is, such that each $\boldsymbol{x}[i]$ can be any nonempty subset of Σ. Observe that every regular string is indeterminate.

The idea of an indeterminate string was first introduced in [12], then studied further in the 1980s as a "generalized string" [1]. Over the last 15 years Blanchet-Sadri has written numerous papers on the properties of "strings with holes" (each $\boldsymbol{x}[i]$ is either a one-element subset of Σ or Σ itself), together with a monograph on the subject [8]; while other authors have studied indeterminate strings in their full generality, together with related algorithms [6, 10, 14, 15, 19, 21–23]. In the specific context of this paper, Voráček & Melichar [24] have done pioneering work on the computation of covers and related structures in generalized strings using finite automata.

For indeterminate strings, equality of letters is replaced by the idea of a "match" [14]: $\boldsymbol{x}[i]$ ***matches*** $\boldsymbol{x}[j]$ (written $\boldsymbol{x}[i] \approx \boldsymbol{x}[j]$) if and only if $\boldsymbol{x}[i] \cap \boldsymbol{x}[j] \neq \emptyset$, while $\boldsymbol{x} \approx \boldsymbol{y}$ if and only if $|\boldsymbol{x}| = |\boldsymbol{y}|$ and corresponding positions in $\boldsymbol{x}$ and $\boldsymbol{y}$ all match. It is important to note that matching is nontransitive: $b \approx \{b, c\} \approx c$, but $b \not\approx c$.

It is [10] that provides the point of departure for our contribution, as we now explain. The ***prefix table*** $\boldsymbol{\pi} = \boldsymbol{\pi}[1..n]$ of $\boldsymbol{x}[1..n]$ is an integer array such that $\boldsymbol{\pi}[1] = n$ and, for every $i \in 2..n$, $\boldsymbol{\pi}[i]$ is the length of the longest substring occurring at position i of $\boldsymbol{x}$ that matches a prefix of $\boldsymbol{x}$. Thus, for our example (1):

$$\begin{array}{rcccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ \boldsymbol{x} = & a & b & a & b & a & b & a & a & b & a \\ \boldsymbol{\pi} = & 10 & 0 & 3 & 0 & 3 & 0 & 1 & 3 & 0 & 1 \end{array}$$

It turns out [7] that the prefix table and the border array are "equivalent" for regular strings; that is, each can be computed from $\boldsymbol{x}$ in linear time, and each can be computed from the other, without reference to $\boldsymbol{x}$, also in linear time. However, for indeterminate strings, this is not true: the prefix table continues to determine all the borders of every prefix of $\boldsymbol{x}$, while the border array, due to the intransitivity of matching, is no longer reliable in identifying borders shorter than the longest one. Consider, for example:

$$\begin{array}{rccc} & 1 & 2 & 3 \\ \boldsymbol{x} = & a & \{a,b\} & b \\ \boldsymbol{\beta} = & 0 & 1 & 2 \end{array}$$

Here $\boldsymbol{x}$ does not have a border of length $\boldsymbol{\beta}[\boldsymbol{\beta}[3]] = 1$; on the other hand, $\boldsymbol{\pi} = 320$ correctly identifies all the borders of every prefix of $\boldsymbol{x}$.

Moreover, it was shown in [10] that every ***feasible*** array — that is, every array $\boldsymbol{y} = \boldsymbol{y}[1..n]$ such that $\boldsymbol{y}[1] = n$ and for every $i \in 2..n$, $\boldsymbol{y}[i] \in 0..n-i+1$ — is a prefix table of some (indeterminate) string. Thus there exists a many-many correspondence between all possible prefix tables and all possible indeterminate strings. Furthermore, [21] describes an algorithm to compute the prefix table of any indeterminate string, while [3] gives an algorithm to compute a lexicographically least indeterminate string corresponding to a given prefix table.

At this point let us discuss our motivation more precisely. First, realize that to exploit the fullest functionality of a border array of an indeterminate string we need to resort to the extended definition of the border array which in fact requires quadratic space [6, 14, 19]: unlike the border array of a regular string, which is a simple array of integers, the border array of an indeterminate string is an array of lists of integers. Here at each position, the list gives all possible borders for that prefix. On the other hand, the prefix array, even for the indeterminate string, remains a simple one-dimensional array, just as for a regular string. It thus becomes of interest to make use of the prefix table rather than the border array whenever possible, in order to extend the scope of computations to indeterminate strings.

In Section 2 of this paper, we describe a linear-time algorithm to compute the cover array $\boldsymbol{\gamma}$ of a regular string $\boldsymbol{x}$ directly from its prefix table $\boldsymbol{\pi}$. Then, Section 3 describes a limited extension of this algorithm to indeterminate strings. Finally, Section 4 outlines future research directions, especially making use of prefix tables to extend the utility and applicability of other data structures to indeterminate strings.

2 Prefix-to-Cover for a Regular String

In this section we describe our basic $\Theta(n)$-time Algorithm `PCR` to compute the cover array $\boldsymbol{\gamma} = \boldsymbol{\gamma}[1..n]$ of a regular string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ directly from its prefix table $\boldsymbol{\pi} = \boldsymbol{\pi}[1..n]$. In fact, as noted in the Introduction, $\boldsymbol{\gamma}$ actually provides all the covers of every prefix of $\boldsymbol{x}$. Central to our algorithm are the following definitions:

Definition 1. *If, for a position $i \in 1..n$, $\boldsymbol{\pi}[i] > 0$, then $R_i = [i, i{+}\boldsymbol{\pi}[i]{-}1]$ is said to be the* ***range*** *at i of* ***length*** *$\boldsymbol{\pi}[i]$; the ranges R_i and $R_{i'}$, $i' > i$, are* ***connected*** *if and only if $i' \leq i{+}\boldsymbol{\pi}[i] < i'{+}\boldsymbol{\pi}[i']$.*

Notably, in what follows, for the sake of brevity, we may slightly abuse the notation $R_i = [i, i{+}\boldsymbol{\pi}[i]{-}1]$ by simply saying $R_i = \boldsymbol{\pi}[i]$.

Definition 2. *Position j in $\boldsymbol{\pi}$ is said to be* ***live*** *at position $i' > j$ if and only if there exists a sequence of $h \geq 1$ connected ranges $R_{i_1}, R_{i_2}, \ldots, R_{i_h}$, each of length at least j, such that $i_1 \leq j{+}1$, $i_h{+}\boldsymbol{\pi}[i_h]{-}1 \geq i'$. Otherwise, j is said to be* ***dead*** *at i'.*

Thus $\boldsymbol{x}[1..n]$ has a cover $\boldsymbol{x}[1..j]$, $j < n$, if and only if j is live at n and the final connected range R_{i_h} satisfies $i_h{+}\boldsymbol{\pi}[i_h]{-}1 = n$.

The strategy of Algorithm `PCR` (Figure 1) is to perform an on-line left-to-right scan of $\boldsymbol{\pi}$, identifying connected ranges R_i. This process may be complex. Within range R_i there may exist two (or more) positions $i_1 > i$ and $i_2 > i_1$ that define ranges R_{i_1} and R_{i_2}, both connected to R_i; of these, `PCR` processes R_i first, followed by R_{i_1}, then, if R_{i_1} and R_{i_2} are connected (they may not be), by R_{i_2}. For example, consider[5]

$$
\begin{array}{rccccccccccccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 \\
\boldsymbol{x} = & b & a & b & a & b & a & b & b & a & b & a & b & a & b & a & b & a & b & a \\
\boldsymbol{\pi} = & 19 & 0 & 5 & 0 & 3 & 0 & 1 & 7 & 0 & 7 & 0 & 7 & 0 & 6 & 0 & 4 & 0 & 2 & 0 \\
\boldsymbol{\gamma} = & 0 & 0 & 0 & 2 & 3 & 4 & 5 & 0 & 0 & 3 & 0 & 5 & 0 & 7 & 0 & 7 & 0 & 7 & 0
\end{array}
\qquad (2)
$$

Here the pairs of ranges (R_8, R_{10}), (R_8, R_{12}) and (R_{10}, R_{12}) are all connected: `PCR` will process positions 8–14 in R_8, followed by 15–16 in R_{10}, then 17–18 in R_{12} and finally position 19 in R_{14}.

Algorithm `PCR` processes each connected range R_i twice, first in left-to-right order, beginning at position $i' = \mathit{lastlim}+1$, where $\mathit{lastlim}$ is the current rightmost position for which $\boldsymbol{\gamma}$ has already been determined, and ending at $i' = \mathit{lim} > \mathit{lastlim}$, the rightmost position in R_i. Corresponding to each i' is the length $j' = i'{-}i{+}1$ of the prefix of R_i (hence also of $\boldsymbol{x}$) that may extend a sequence of covering substrings of length j'. In order to determine whether or not j' is live at i', `PCR` maintains an array $\mathit{maxlive}[1..n]$, using the following values:

[5] Thanks to Alice Heliou, Laboratoire d'Informatique de l'École Polytechnique, Palaiseau, France.

```
procedure PCR (π, γ)
γ[1..n] ← 0^n; maxlive[1..n] ← 0^n
lastlim ← 1; i ← 2
while lastlim < n do
    j ← π[i]
    if j = 0 then
     ▷ No range extends beyond lastlim, so 1, 2, ..., i−1 are all dead.
        if i > lastlim then
            maxlive[i−1] ← −1; lastlim ← i
    else
        lim ← i+j−1
        if lim > lastlim then
            j' ← (lastlim+1) − i
         ▷ Initial setting of maxlive and γ.
            for i' ← lastlim+1 to lim do
                j' ← j'+1
                if (maxlive[j'] = 0 and i' ≤ 2j')
                or maxlive[j'] ≥ i'−j' then
             ▷ j' is a cover of x[1..i'].
                    maxlive[j'] ← i'; γ[i'] ← j'
                else
             ▷ j' is ruled out as a cover.
                    maxlive[j'] ← −1
         ▷ Reset maxlive and γ in case of multiple covers.
            for i' ← lim downto lastlim+1 do
                j'' ← γ[j']
             ▷ A cover of x[1..j'] is also a cover of x[1..i'].
                while j'' > 0 and 0 < maxlive[j''] < i' do
                    maxlive[j''] ← i'; γ[i'] ← max(γ[i'], j'')
                    j'' ← γ[j'']
                j' ← j'−1
            lastlim ← lim
    i ← i+1
```

Fig. 1. Compute the cover array $\boldsymbol{\gamma}$ of a regular string $\boldsymbol{x}$ from its prefix table $\boldsymbol{\pi}$.

$$maxlive[j'] = \begin{cases} 0 : \text{initial setting: position } j' \text{ not yet considered} \\ i' : j' \text{ live at } i'\text{: } \boldsymbol{x}[1..i'] \text{ covered by } \boldsymbol{x}[1..j'] \\ -1 : j' \text{ is (permanently) dead} \end{cases}$$

However, it can happen that *maxlive* and $\boldsymbol{\gamma}$ are not correctly set by the left-to-right scan of R_i:

Definition 3 ([16]). *In the cover array $\boldsymbol{\gamma}$, if there exists an integer $k \geq 1$ and positions $i > j > 0$ such that $\boldsymbol{\gamma}^k[i] = j$, then j is said to be the $k^{\boldsymbol{th}}$* ***ancestor*** *of i in $\boldsymbol{\gamma}$. Thus the cover array determines a* ***cover tree****.*

It may be that $\boldsymbol{\gamma}[i']$ is set to zero because j' is dead at i', even though an ancestor of j' in the cover tree is live at i'; on the other hand, when $\boldsymbol{\gamma}[i'] = j'$, so that ancestors of j' may also be live at i', the *maxlive* values of the ancestors may need to be adjusted. Thus a second right-to-left scan of R_i is required, in order to ensure that these updates are correct.

For example, in (2), we need to ensure that $maxlive[5] = maxlive[3] = 18$, since both 5 and 3 are live ancestors of 7. A more subtle example is given in (3), where at position 19 we need to recognize that both 5 and 3 are live, even though 7 is dead, so that later, at position 22, we can recognize that 3 is live:

$$\begin{array}{rcccccccccccccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 \\ \boldsymbol{x} = & b & a & b & a & b & a & b & b & a & b & a & b & b & a & b & a & b & a & b & b & a & b \\ \boldsymbol{\pi} = & 22 & 0 & 5 & 0 & 3 & 0 & 1 & 5 & 0 & 3 & 0 & 1 & 7 & 0 & 5 & 0 & 3 & 0 & 1 & 3 & 0 & 1 \\ \boldsymbol{\gamma} = & 0 & 0 & 0 & 2 & 3 & 4 & 5 & 0 & 0 & 3 & 0 & 5 & 0 & 0 & 3 & 0 & 5 & 0 & 5 & 0 & 0 & 3 \end{array} \tag{3}$$

Consider also

$$\begin{array}{rcccccccccccccccccccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 \\ \boldsymbol{x} = & b & a & b & a & b & a & b & b & a & b & a & b & a & b & a & b & b & a & b & a & b & b & a & b & a & b & a & b \\ \boldsymbol{\pi} = & 22 & 0 & 5 & 0 & 3 & 0 & 1 & 7 & 0 & 7 & 0 & 5 & 0 & 3 & 0 & 1 & 5 & 0 & 3 & 0 & 1 & 7 & 0 & 5 & 0 & 3 & 0 & 1 \\ \boldsymbol{\gamma} = & 0 & 0 & 0 & 2 & 3 & 4 & 5 & 0 & 0 & 3 & 0 & 5 & 0 & 7 & 0 & 7 & 0 & 0 & 3 & 0 & 5 & 0 & 0 & 3 & 0 & 5 & 0 & 5 \end{array} \tag{4}$$

Thus, using n additional words of storage and a double scan of each connected range, Algorithm `PCR` is able to compute $\boldsymbol{\gamma}$. The time requirement is $\Theta(2n)$ plus the time required by the internal **while** loop; this loop updates $maxlive[j']$ at most once for each ancestral position j' in the range, thus requiring a total $O(n)$ time overall. Hence we have the following result:

Theorem 1. *Given the prefix table $\boldsymbol{\pi}$ of a regular string $\boldsymbol{x} = \boldsymbol{x}[1..n]$, Algorithm* `PCR` *correctly computes the cover array $\boldsymbol{\gamma}$ of $\boldsymbol{x}$ in $\Theta(n)$ time using an additional n integers of space.*

$$\begin{array}{rccccccccccccccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 \\ \boldsymbol{x} = & a & b & a & a & b & a & b & a & a & b & a & a & b & a & b & a & a & b & a & b & a & b & a \\ \pi = & 23 & 0 & 1 & 3 & 0 & 6 & 0 & 1 & 11 & 0 & 1 & 3 & 0 & 8 & 0 & 1 & 3 & 0 & 3 & 0 & 3 & 0 & 1 \\ \gamma = & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 3 & 0 & 5 & 6 & 0 & 5 & 6 & 0 & 8 & 9 & 10 & 11 & 0 & 8 & 0 & 3 \end{array} \quad (5)$$

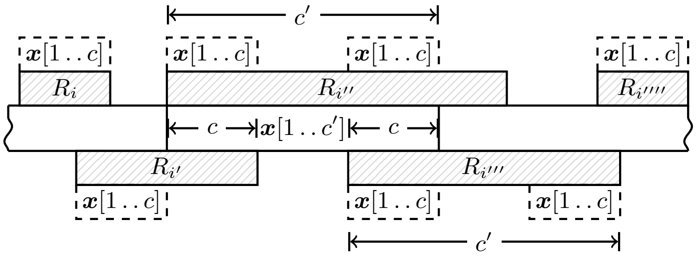

Fig. 2. Showing two covers from $\gamma(\boldsymbol{x})$, $\boldsymbol{x} = abaababaabaababaababababa$ (5)

Given a string $\boldsymbol{x} = abaababaabaababaababababa$ (5), the figure shows two covers from $\gamma(\boldsymbol{x})$, namely $c = 3$ and $c' = 8$, and also shows selected ranges from the prefix array $\pi(\boldsymbol{x})$ that explicitly participate in the generation of these covers: the ranges are $R_i = \pi[4] = 3,\ R_{i'} = \pi[6] = 6,\ R_{i''} = \pi[9] = 11,\ R_{i'''} = \pi[14] = 8,\ R_{i''''} = \pi[21] = 3$.

3 Extensions to Indeterminate Strings

It turns out that for indeterminate strings there are two natural analogues of the idea of "cover".

Definition 4. *A string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is said to have a **sliding cover** of length κ if and only if*

(a) $\boldsymbol{x}$ has a suffix $\boldsymbol{v}$ of length $|\boldsymbol{v}| = \kappa$; and
(b) $\boldsymbol{x}$ has a proper prefix $\boldsymbol{u}$, $|\boldsymbol{u}| \geq |\boldsymbol{x}| - \kappa$, with suffix $\boldsymbol{v'} \approx \boldsymbol{v}$; and
(c) either $\boldsymbol{u} = \boldsymbol{v'}$ or else $\boldsymbol{u}$ has a cover of length κ.

A sliding cover requires that adjacent or overlapping substrings of $\boldsymbol{x}$ match, but the nontransitivity of matching leaves open the possibility that nonadjacent elements of the cover do not match. For example,

$$\boldsymbol{x} = \{a,b\}c\{a,c\}\{a,c\}ca \quad (6)$$

has a sliding cover of length $\kappa = 2$ because $\{a,b\}c \approx \{a,c\}\{a,c\} \approx ca$, even though $\{a,b\}c \not\approx ca$.

However, note that the very concept of “regularity of a string” in some sense breaks down when we consider the concept of a sliding cover: now the “cover” need not actually “match” the area it is covering. In fact, the above concept even allows for a string to be a cover of an indeterminate string without being a substring of the latter at all! This motivates the idea of a ***rooted cover*** of length κ, where every covering substring is required to match, not the preceding entry in the cover, but rather the prefix of $\boldsymbol{x}$ of length κ. A rooted cover is defined simply by changing “suffix” to “prefix” in part (b) of Definition 4. The example string (6) has no rooted cover, but the string $\boldsymbol{x'} = \{a,b\}c\{a,c\}\{a,c\}ac$ has both a sliding cover and a rooted cover of length 2. Notably, in the literature, the concept of rooted cover is in fact used as the cover for an indeterminate string [6].

3.1 Computing Rooted Covers

In this section we describe Algorithm `PCInd` (Fig. 3) to compute the set of rooted covers Γ of a given indeterminate string $\boldsymbol{x} \in \Sigma^n$ directly from its prefix table. As will be shown below, the algorithm runs in linear time on average and $O(n^2)$ time in the worst case.

Algorithm `PCInd` maintains a list $\mathcal{L}$ to store the candidate rooted covers. The algorithm also maintains an auxiliary push-down store $\mathcal{D}$, which stores the list of dead covers at each iteration $i \in [2..n]$. The push-down store $\mathcal{D}$ will be used for marking the dead covers so as to delete them at the end of each iteration. Lastly, in order to determine whether or not the cover of length v is live at position i, the algorithm maintains an array $maxlive[1..n]$ the same as in Algorithm `PCR`.

Exploiting the fact that the rooted cover of an indeterminate string $\boldsymbol{x}$ is also a border of it, the algorithm starts by identifying the set of candidate (rooted) covers as defined below.

Definition 5. *Let $\boldsymbol{x} \in \Sigma^n$ and let $\pi[1..n]$ be its prefix array. Then the set of candidate (rooted) covers $\mathcal{L}$ of the whole string $\boldsymbol{x}$ is:*

$$\mathcal{L} \subseteq \pi : \text{ where } \pi[i] + i - 1 = n \text{ for } 2 \le i \le n \qquad (7)$$

To populate the list of candidate covers, we start by computing the value $max = \max(\pi[2..n])$. Then the algorithm initializes the list $\mathcal{L}$ with the filtered entries from the set $\{1, 2, ..., max\}$, such that $\mathcal{L}$ will only store the values that satisfies $y[i] + i - 1 = n$ for $i \in [2..n]$.

During the execution of the main **for** loop, at each position $i \in [2..n]$. The algorithm tests, for each candidate cover v in list $\mathcal{L}$, whether or not v is active. Based on the result of this test the algorithm appropriately updates the corresponding entry in the $maxlive$ array and marks the dead covers at position i, by storing those in $\mathcal{D}$ which will be deleted at the end of each iteration using a **while** loop.

After computing the array $maxlive$ (at the end of the main **for** loop), we can easily identify and report the set of rooted covers of the whole string $\boldsymbol{x}$ simply

```
procedure PCInd(π, Γ)
    Γ ← φ;  L ← φ;  maxlive[1..n] ← 0^n
    max ← max(π[2..n])
    ▷ fill the list L with the candidate covers from {1, 2, ..., max}
    for i ← 1 to max do
        ▷ consider only border values
        if π[i] + i − 1 = |s| then
            L ←+ i
    for i ← 2 to n do
        ▷ D stores list of dead covers at position i
        D ← φ
        for all (v ∈ L) do
            ▷ skip values of v > π[i]
            if (v > π[i]) then
                break
            t ← i + v − 1
            if ((maxlive[v] = 0 and t ≤ 2 ∗ v)
                or (maxlive[v] ≥ t − v)) then
                ▷ cover v is still live
                maxlive[v] = t
            else
                ▷ cover v is dead
                maxlive[v] = −1
                ▷ mark cover v for deletion
                push(D) ← v
        ▷ remove the dead covers from L
        while top(D) ≠ ∅ do
            r ← pop(D)
            L ←− r
    ▷ report the rooted covers
    for i ← 1 to n do
        if maxlive[i] = n then
            Γ ←+ i
```

Fig. 3. Compute all rooted covers of indeterminate string from its prefix array.

by finding all the entries in the array *maxlive* that have the value n (i.e., all entries of the list of candidate covers that are still active).

A final note regarding the use of the push-down store $\mathcal{D}$ is in order. The standard approach, when the programming language in use allows it, is to delete some elements from a list while iterating through it. This can be done either: (1) by iterating backwards through the list and then deleting within the **for** loop, or (2) by identifying all items that need to be deleted and marking them with a flag (in the first iteration), then (in the second iteration) removing all those items which are flagged for deletion. However, in both cases (1) and (2), the algorithm must loop through all the items in the list $\mathcal{L}$ after each iteration. Alternatively, keeping track of the items to remove in another list (e.g., in $\mathcal{D}$) and then, after all items have been processed, enumerating the remove list ($\mathcal{D}$) and removing each item from the list of candidate covers ($\mathcal{L}$) requires only looping through $\mathcal{D}$.

3.2 Analysis

Finding the value max in $\pi[2..n]$ can be done with a simple linear scan of the array π. Computing the list $\mathcal{L}$ of candidate covers can be done in $O(n)$ time. The main **for** loop will be executed exactly n times.

Within the loop the checking of the condition whether a cover is active or not can be done in constant time for a particular value and hence the total testing of *live* or *dead* for all candidate covers requires time proportional to $|\mathcal{L}|$, which is $O(n)$ in the worst case. Note that the list $\mathcal{L}$ tends to get smaller and smaller as the iteration continues, because we keep removing dead covers from it after each iteration. However, the complexity remains $O(n)$ in the worst case (e.g., $\boldsymbol{x} = a^n$).

Turning our attention to the **while** loop at the end of each iteration of the main **for** loop, the processing of $\mathcal{D}$ to remove the dead covers also requires time proportional to $\mathcal{D}$, thus $O(n)$ in the worst case since the total number of covers is bounded by n. We conclude that the worst-case time requirement for the main **for** loop is $O(n^2)$. The final **for** loop to report the list of rooted covers requires time proportional to $|maxlive|$ which is $O(n)$. The algorithm requires linear extra space to store the lists $maxlive$, $\mathcal{L}$ and $\mathcal{D}$. So we have the following result:

Theorem 2. *Given the prefix table π of an indeterminate string $\boldsymbol{x} = \boldsymbol{x}[1..n]$, Algorithm* `PCInd` *correctly computes the set of rooted covers of the whole string of $\boldsymbol{x}$ in $O(n^2)$ time and linear space.*

Finally, Bari et. al. [6] proved that the expected number of borders of an indeterminate string is bounded by a constant. Since, in the beginning of Algorithm `PCInd` we include only the borders in $\mathcal{L}$, this means that the size of the list $\mathcal{L}$ and also $\mathcal{D}$ is bounded by a constant. Therefore, based on the analysis presented above we can conclude that Algorithm `PCInd` runs in linear time on average.

3.3 An Illustrative Example

Suppose $\pi = \{12, 3, 2, 1, 1, 7, 6, 1, 0, 3, 0, 1\}$. We have $max = 7$. The simulation of the algorithm is shown in Fig. 4. The algorithm initializes the set $\mathcal{L}$ with the set of candidate covers. Hence, we have $\mathcal{L} = \{1, 3, 6, 7\}$. At iteration $i = 6$, we can see that cover 3 becomes non-active, so the value $maxlive[3]$ is set to -1 and the cover 3 is removed from the set of candidate covers. Similarly, at iteration $i = 10$, the cover 1 becomes non-active, so the value $maxlive[1]$ is set to -1 and the cover 1 is removed from the set of candidate covers. After computing the array *maxlive*, the list of rooted covers can be identified as all the positions i in *maxlive* where $maxlive[i] = n$. So the covers are 6 and 7 since $maxlive[6] = 12$ and $maxlive[7] = 12$. We have $\Gamma = \{6, 7\}$.

i	*maxlive*	$\mathcal{L}$
2	$\{2, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$	$\{1, 3, 6, 7\}$
3	$\{3, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$	$\{1, 3, 6, 7\}$
4	$\{4, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$	$\{1, 3, 6, 7\}$
5	$\{5, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$	$\{1, 3, 6, 7\}$
6	$\{6, 0, -1, 0, 0, 11, 12, 0, 0, 0, 0, 0\}$	$\{1, 6, 7\}$
7	$\{7, 0, -1, 0, 0, 12, 12, 0, 0, 0, 0, 0\}$	$\{1, 6, 7\}$
8	$\{8, 0, -1, 0, 0, 12, 12, 0, 0, 0, 0, 0\}$	$\{1, 6, 7\}$
9	$\{8, 0, -1, 0, 0, 12, 12, 0, 0, 0, 0, 0\}$	$\{1, 6, 7\}$
10	$\{-1, 0, -1, 0, 0, 12, 12, 0, 0, 0, 0, 0\}$	$\{6, 7\}$
11	$\{-1, 0, -1, 0, 0, 12, 12, 0, 0, 0, 0, 0\}$	$\{6, 7\}$
12	$\{-1, 0, -1, 0, 0, 12, 12, 0, 0, 0, 0, 0\}$	$\{6, 7\}$

Fig. 4. The running values of Algorithm `PCInd` for a given string with prefix array $\pi = \{12, 3, 2, 1, 1, 7, 6, 1, 0, 3, 0, 1\}$

3.4 The experiment

To get an idea of how the algorithm behaves in practice, we have implemented Algorithm `PCInd` and conducted a simple experimental study. The experiments have been carried out on a Windows Server 2008 R2 64-bit Operating System, with Intel(R) Core(TM) i7 2600 processor @ 3.40GHz having an installed memory (RAM) of 8.00 GB. The algorithm have been implemented in *C#* language using Visual Studio 2010.

We have run Algorithm `PCInd` on a set of 100 randomly generated prefix arrays for each length $n \in \{100, 200, \ldots, 100, 000\}$ (averaged over 100 runs for each length) and counted the average number of executions of the inner loop of the algorithm. The resulting graph (Fig. 5) shows the average complexity of Algorithm `PCInd` fluctuating around n. Note that the values n^2 in the graph are scaled down by $10, 000$ (i.e., the curves are showing $n^2/10, 000$) to have a better view of the curves. The results show that the run time of the algorithm is close to linear confirming the average case time complexity of $O(n)$.

4 Future Directions

There are several data structures related to the cover array whose computation may now be contemplated in the context of indeterminate strings. For example, a recent paper [13] introduces new forms of “enhanced” cover array that are

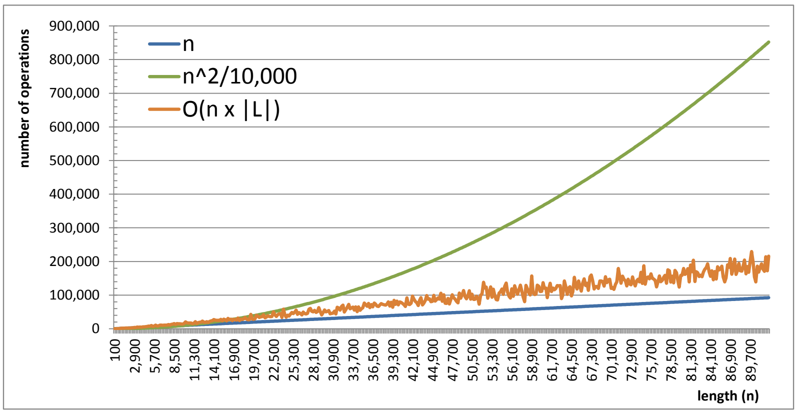

Fig. 5. The average running time of the Algorithm `PCInd`.

efficiently computed using the border array; using the cover array instead would open the way for computation of variants of these structures also for indeterminate strings. Similarly, another recent paper [11] proposes efficient algorithms for the computation of "seed" arrays (a ***seed*** of a string $\boldsymbol{x}$ is a cover of some superstring of $\boldsymbol{x}$) — these algorithms also may be similarly extended.

References

1. Karl Abrahamson, **Generalized string matching**, *SIAM J. Computing 16–6* (1987) 1039–1051.
2. Alfred V. Aho, John E. Hopcroft & Jeffey D. Ullman, *The Design & Analysis of Computer Algorithms*, Addison-Wesley (1974).
3. Ali Alatabbi, M. Sohel Rahman, & W. F. Smyth, **Inferring an indeterminate string from a prefix graph**, *J. Discrete Algorithms* (2014), doi:10.1016/j.jda.2014.12.006.
4. Alberto Apostolico & Andrzej Ehrenfeucht, *Efficient Detection of Quasiperiodicities in Strings*, Tech. Report No. 90.5, The Leonardo Fibonacci Institute, Trento, Italy (1990).
5. Alberto Apostolico, Martin Farach & Costas S. Iliopoulos, Optimal superprimitivity testing for strings, *Inform. Process. Lett. 39-1* (1991) 17-20.
6. Md. Faizul Bari, Mohammad Sohel Rahman & Rifat Shahriyar, **Finding All Covers of an Indeterminate String in O(n) Time on Average**, *Stringology* (2009) 263–271.
7. Widmer Bland, Gregory Kucherov & W. F. Smyth, **Prefix table construction & conversion**, *Proc. 24th IWOCA*, Springer Lecture Notes in Computer Science LNCS 8288 (2013) 41–53.
8. Francine Blanchet-Sadri, *Algorithmic Combinatorics on Partial Words*, Chapman & Hall/CRC (2008) 385 pp.
9. D. Breslauer, An on-line string superprimitivity test, *Inform. Process. Lett. 44-6* (1992) 345-347.

10. Manolis Christodoulakis, P, J. Ryan, W. F. Smyth & Shu Wang,. **Indeterminate strings, prefix arrays & undirected graphs**, *CoRR abs/1406.3289* (2014).
11. Michalis Christou, Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Bartosz Szreder & Tomasz Walen, **Efficient seeds computation revisited**, *Proc. 22nd Annual Symp. Combinatorial Pattern Matching*, Raffaele Giancarlo & Giovanni Manzini (eds.), Lecture Notes in Computer Science, LNCS 6661, Springer-Verlag (2011) 350–363.
12. Michael J. Fischer & Michael S. Paterson, **String-matching and other products**, *Complexity of Computation, Proc. SIAM-AMS 7* (1974) 113-125.
13. Tomáš Flouri, C. S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Simon J. Puglisi, W. F. Smyth & Wojciech Tyczyński, **Enhanced string covering**, *Theoret. Comput. Sci. 506* (2013) 102–114.
14. Jan Holub & W. F. Smyth, **Algorithms on indeterminate strings**, *Proc. 14th Australasian Workshop on Combinatorial Algs.* (2003) 36–45.
15. Jan Holub, W. F. Smyth & Shu Wang, **Fast pattern-matching on indeterminate strings**, *J. Discrete Algorithms 6–1* (2008) 37–50.
16. Yin Li & W. F. Smyth, **Computing the Cover Array in Linear Time**, *Algorithmica 32–1* (2002) 95–106.
17. Dennis Moore & W. F. Smyth, An optimal algorithm to compute all the covers of a string, *Inform. Process. Lett. 50* (1994) 239-246.
18. Dennis Moore & W. F. Smyth, Correction to: An optimal algorithm to compute all the covers of a string, *Inform. Process. Lett. 54* (1995) 101-103.
19. Sumaiya Nazeen, M. Sohel Rahman & Rezwana Reaz, **Indeterminate string inference algorithms**, *J. Discrete Algorithms 10* (2012) 23–34.
20. Bill Smyth, *Computing Patterns in Strings*, Pearson Addison-Wesley (2003) 423 pp.
21. W. F. Smyth & Shu Wang, **New perspectives on the prefix array**, *Proc. 15th String Processing & Inform. Retrieval Symp.*, Springer Lecture Notes in Computer Science LNCS 5280 (2008) 133–143.
22. W. F. Smyth & Shu Wang, **A new approach to the periodicity lemma on strings with holes**, *Theoret. Comput. Sci. 410–43* (2009) 4295–4302.
23. W. F. Smyth & Shu Wang, **An adaptive hybrid pattern-matching algorithm on indeterminate strings**, *Internat. J. Foundations of Computer Science 20–6* (2009) 985–1004.
24. M. Voráček & B. Melichar, **Searching for regularities in generalized strings using finite automata**, *Proc. Internat. Conf. on Numerical Analysis & Applied Maths.* Wiley-VCH (2005).