AIMS Medical Science

AIMS Medical Science, 4(3): 261–273
DOI: 10.3934/medsci.2017.3.261
Received 25 April 2017,
Accepted 22 June 2017,
Published 13 July 2017

http://www.aimspress.com/journal/medicalScience

Research article

The Role of The Prefix Array in Sequence Analysis: A Survey

Frantisek Franek [1, 2], W. F. Smyth [1, 2, 3, *] and Xinfang Wang [2]

[1] Department of Computing and Software, McMaster University, Hamilton, Canada
[2] School of Computational Science and Engineering, McMaster University, Hamilton, Canada
[3] School of Engineering and Information Technology, Murdoch University, Perth, Australia

*** Correspondence:** E-mail: smyth@mcmaster.ca; Tel: +1-905-525-9140 ext 23436

Abstract: The prefix array was apparently first computed and used algorithmically in 1984, playing a pivotal role in an optimal algorithm to determine all the tandem repeats in a given (DNA or protein) sequence. However, it is especially since the turn of the 21st century that applications of the prefix array to fundamental sequencing problems have been recognized. An important aspect of this expanding role has been the recognition that the prefix table and the border array are "equivalent" data structures — that is, one can be computed from the other in linear time. Since the border array in turn specifies all the periods of every prefix of the sequence, the prefix array thus turns out to be a structure of central importance. In this paper we survey important applications of the prefix array — in particular to approximate string matching under Hamming distance, as well as to the computation of covers and enhanced covers — and show how, unlike border array algorithms, these are extendible to sequences containing "don't-care" or indeterminate letters such as $\{a, c\}$ or $\{g, t\}$. This extension leads to a surprising correspondence between prefix arrays and undirected graphs that seems likely to be a fertile source of new insights in future. We conclude with an overview of sequencing problems that the authors believe can be handled using prefix array technology.

Keywords: string; sequence; indeterminate string; prefix array; sequence analysis; algorithm

1. Introduction

DNA and protein sequences are special cases of what mathematicians call "words" and computer scientists call "strings" — that is, sequences of "letters" drawn from some alphabet. Thus, in the case of DNA, the alphabet consists of abbreviations {*a, c, g, t*} of the four nucleotides that characterize all known life. In this paper, because we are discussing computer processing of these objects, we will usually refer to them as strings.

For more than 60 years, biologists, mathematicians and computer scientists have studied biological sequences in an effort to understand the patterns (or "regularities" [1,2]) in them that somehow yield the astonishing complexity and diversity of life on earth. Due to the great length of these sequences, the computer has become a necessary tool of "sequence analysis" and, as time goes by, a collection of data structures has been developed to make string processing more efficient, thus to permit ever more sophisticated calculations to be carried out. In this collection researchers will for example recognize [1]:

- the ubiquitous border array or failure function [3], famously used in the KMP pattern-matching algorithm [4], and since then certainly in hundreds of string algorithms;
- the TRIE data structure for storing multiple strings [5];
- a very special TRIE, the suffix tree [6–8], with all its many variants and uses [9];
- the suffix array [10,11] that was known for 13 years before three $O(n)$ time algorithms were suddenly found to compute it [12–15], and that now, based on an even more efficient construction algorithm [16] and greatly expanded applications [17], has largely replaced the suffix tree.

In this survey we focus on the prefix array, apparently discovered in 1984, ignored by stringologists for 15 years, but recently found to be central to several interesting applications. As we shall see, the prefix array is more or less "equivalent" to the border array, but with the advantage that it can be used on sequences whose letters may be ambiguous ("indeterminate") — for example, a nucleotide {*a, c*} that may be either adenine or cytosine.

Section 2 provides the basic definitions required to explain the algorithms outlined in later sections, and also discusses the correspondence between the border and prefix arrays, explaining the advantage of the latter in algorithms on strings that contain indeterminate letters. Section 3 shows how the prefix array can be applied to determining the Hamming distance between strings containing indeterminate letters. In Section 4 we introduce the idea of a cover of a string, and again show the relevance of the prefix array to computation of the cover, also again in indeterminate strings. Section 5 discusses ideas for future applications.

2. Preliminaries

2.1 Basic Definitions

Given an integer $n \geq 0$, a ***string x*** is a sequence of n ***letters*** drawn from a finite set Σ called the ***alphabet***, where we write $\sigma = |\Sigma|$. We treat $\boldsymbol{x}$ as an array $\boldsymbol{x}[1..n]$, and if $n = 0$, we write $\boldsymbol{x} = \varepsilon$, the ***empty string***. Then $n = |\boldsymbol{x}|$ is said to be the ***length*** of $\boldsymbol{x}$. Of course, in bioinformatics applications, we routinely process strings whose length n is in the billions.

If $\boldsymbol{x} = \boldsymbol{uvw}$, then $\boldsymbol{u}$ (respectively, $\boldsymbol{v}$, $\boldsymbol{w}$) is said to be a ***prefix*** (respectively, ***substring***, ***suffix***) of $\boldsymbol{x}$ — in each case proper if $|\boldsymbol{u}| < n$ (respectively, $|\boldsymbol{v}| < n$, $|\boldsymbol{w}| < n$). If $\boldsymbol{u}$ is a substring of $\boldsymbol{x}$, then $\boldsymbol{x}$ is a ***superstring*** of $\boldsymbol{u}$. If $\boldsymbol{u}$ is both a proper prefix and a proper suffix of $\boldsymbol{x}$, then $\boldsymbol{u}$ is said to be a ***border*** of $\boldsymbol{x}$ — note that ε is a border of every nonempty string. If $\boldsymbol{x} = \boldsymbol{u}^k$ for some nonempty $\boldsymbol{u}$ and integer $k > 1$, we say that $\boldsymbol{x}$ is a ***repetition*** or ***tandem repeat*** [2] — when $k = 2$, a ***square***. If $\boldsymbol{x}$ is not a repetition, it is said to be ***primitive***. If $\boldsymbol{x} = \boldsymbol{u}^k\boldsymbol{u'}$ for some nonempty $\boldsymbol{u}$, integer $k > 0$, and $\boldsymbol{u'}$ a proper prefix of $\boldsymbol{u}$, then $\boldsymbol{x}$ is said to have period $|\boldsymbol{u}|$. The following results are well known (see for example [18]):

Observation 1

a) *String* $\boldsymbol{x}$ *of length n has period p if and only if it has a border of length n-p.*

b) *Suppose* $\boldsymbol{x}' = \boldsymbol{x}[1..\beta_1]$ *is a border of* $\boldsymbol{x}$ *and suppose* $\beta_2 \in 0..\beta_1 - 1$ *is an integer* [3]. *Then* $\boldsymbol{x''} = \boldsymbol{x}[1..\beta_2]$ *is a border of* $\boldsymbol{x}'$ *if and only if* $\boldsymbol{x''}$ *is a border of* $\boldsymbol{x}$.

The ***border array*** $\beta_{\boldsymbol{x}} = \beta[1..n]$ gives for every $i \in 1..n$ the length $\beta_{\boldsymbol{x}}[i]$ of the longest border of $\boldsymbol{x}[1..i]$; a well-known algorithm [3,4] computes $\beta_{\boldsymbol{x}}$ in $O(n)$ time. From Observation 1(b) it follows that $\beta_{\boldsymbol{x}}$ specifies all the borders of *every* nonempty prefix of $\boldsymbol{x}$. See Figure 1.

So far we have made the implicit assumption that the letters of $\boldsymbol{x}$ are single elements of Σ, an assumption that in many contexts may not be justified. For example, as mentioned earlier, it may be that for some i, the letter $\boldsymbol{x}[i]$ is ambiguous, thus best represented by $\{a, c\}$ (either a or c) or perhaps $\{a, c, g, t\}$ (any element of Σ). The next subsection discusses issues arising in such cases, in particular the limitations of the border array, and introduces an "equivalent" data structure, the prefix array, which is more widely applicable.

4

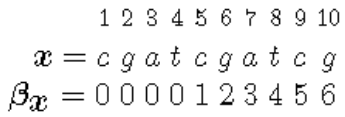

Figure 1. A string *x* of length *n* = 10 on alphabet Σ = {*c*, *g*, *a*, *t*} with proper prefix *cga*, proper substrings *tc* and *gat*, proper suffix *atcg*. Since *x* is not a repetition, it is therefore primitive. ε, *cg* and *cgatcg* are the borders of *x*, that accordingly has periods *n*, *n*−|*cg*| = 8 and *n*−|*cgatcg*| = 4.

2.2 Indeterminate String, Border Array, Prefix Array

A letter λ drawn from alphabet $\Sigma = \{f_1, f_2, ..., f_\sigma\}$ is said to be regular if $\lambda = f_j$ for some $j \in 1..\sigma$; otherwise, if $\lambda = \Sigma_k$, a subset of Σ of size $k > 1$, then λ is said to be ***indeterminate***. We say that two letters λ_1 and λ_2 match, written $\lambda_1 \approx \lambda_2$, if and only if $\lambda_1 \cap \lambda_2 \neq \emptyset$; thus regular letters match if and only if they are equal, while for indeterminate letters this is not true. In fact, "match" is in general nontransitive, as the following example shows:

$\lambda_1 = f_1, \lambda_2 = f_2, \lambda_3 = \{f_1, f_2\} \Rightarrow \lambda_1 \approx \lambda_3 \approx \lambda_2$ but $\lambda_1 \not\approx \lambda_2$.

Similarly, a string ***x*** on Σ is said to be ***regular*** if every letter $\boldsymbol{x}[i]$, $1 \leq i \leq n$, is regular; otherwise ***indeterminate***. But note that a string — for example, $\boldsymbol{x} = \{c, g\}, a, t, \{c, g\}$ — may be indeterminate but at the same time give rise only to transitive matches; such strings are called ***essentially regular***. Strings with letters restricted to either single elements f*j* of Σ or else Σ itself (called a ***hole*** or ***don't care*** letter) were introduced in [19] and have been intensively studied as partial words since 2003 by Blanchet-Sadri (see [20]). In the 1980s Abrahamson [21] dealt with a form of indeterminacy (which he called "generalized string matching"); in this century, beginning with [22], there has been continued interest in indeterminate strings — for example, see [23,24].

In order to be useful for indeterminate strings, the border needs to be redefined in terms of matching rather than equality: a proper prefix ***u*** of ***x*** is a border of ***x*** if and only if ***u*** ≈ ***u***', where ***u***' is a suffix of ***x***. But then Observation 1(b) no longer holds, as Figure 2 shows, and so the border array loses its usefulness as a compact representation of all the borders of every prefix of a string.

We now introduce the ***prefix array*** π_x of a string ***x***: for every $i \in 1..n$, $\pi_x[i] = u$, where u is the largest integer such that $\boldsymbol{x}[i..i + u - 1] \approx \boldsymbol{x}[1..u]$, thus also providing the critical information that, for $i + u \leq n$, $\boldsymbol{x}[i + u] \not\approx \boldsymbol{x}[u + 1]$. Again see Figure 2.

$$
\begin{array}{rcccc}
 & 1 & 2 & 3 & 4 \\
x = & \{a,g\} & \{a,t\} & \{c,g\} & \{c,t\} \\
\beta_x = & 0 & 1 & 1 & 2 \\
\pi_x = & 4 & 1 & 2 & 0
\end{array}
$$

Figure 2. *x'*= {*a*, *g*} {*a*, *t*} (or {*c*, *g*} {*c*, *t*}) is a border of x; but neither *x"* = {*a*, *g*} nor {*a*, *t*} is a border of *x*[3..4], and neither *x"* = {*c*, *g*} nor {*c*, *t*} is a border of *x*[1..2]. However π_x correctly identifies all the borders of every prefix of *x*: π[3..4] = 20 ⇒ *x* has border only of length 2; π[3] > 0 and π[2[< 2 ⇒ *x*[1..3] has border only of length 1; π[2] > 0 ⇒ *x*[1..2] has a border of length 1.

The prefix array was apparently first used in Main and Lorentz's algorithm [18,25] to compute all the tandem repeats in a string, but only in this century was it identified as an important data structure, the *table des préfixes* [26,27], where it was also for the first time observed that (on regular strings) the border array and the prefix array could be computed from each other in O(n) time, and so were in some sense "equivalent". However, in [28] it was shown that, unlike the border array, the prefix array retained its properties on indeterminate strings, on which it could be used for pattern-matching [4] and other applications (see Figure 2). Also in [28] a compressed form of the prefix array was introduced, making use of the fact that entries in πx are usually zero — $\pi x[i] = 0 \Leftrightarrow x[i] \not\approx x[1]$. Linear-time algorithms for computing the border and prefix arrays of a string *x*, as well as for converting from one array to the other, were described and analyzed in [28,29].

We note also [30] that the prefix array gives rise to a simple and efficient pattern-matching algorithm: given a string ***x*** and a pattern ***u***, compute the prefix array of ***w*** = ***u***\$***x***, where \$ is a letter not occurring in ***x***. Then the positions *i* such that $\pi_w[i] = |\boldsymbol{u}|$ identify the occurrences of ***u*** in ***x***. (In Section 3 we shall see a variant of this idea used to do pattern-matching with *k* mismatches.)

In [31] a linear-time algorithm was described that, given an integer array $I = I[1..n]$, determined in $O(n)$ time whether *I* was a prefix array of some regular string, and, if so, found a lexicographically least string ***x*** for which $\pi_x = I$. This work was extended to indeterminate strings in [32]. In [33] the ***feasible array*** $y = y[1..n]$ was introduced, with $y[1] = n$ and, for every $i \in 2..n$, $i \leq I + y[i] \leq n + 1$. It was then shown that every feasible array is the prefix array of some (indeterminate) string, while of course every prefix array must be feasible. Also in [33], a ***prefix graph*** with interesting properties was defined corresponding to every feasible array; this has led to new theoretical insights [34,35].

Recently the range of application of the prefix array was extended even further with the publication of a paper [36] describing a prefix array for ***parameterized strings*** — that is, strings on two alphabets, a "variable" alphabet of the usual kind together with a special "fixed" alphabet of

specified parameters. In a biological context, for example, we might fix parameters $\boldsymbol{A} = aca$ and $\boldsymbol{B} = gtg$, then search a string $\boldsymbol{x}$ for all occurrences of $\boldsymbol{A} * \boldsymbol{B}$, where * indicates any sequence of variables drawn from $\{a, c, g, t\}$. The original application of parameterized strings was the identification of cloned computer code: a program on a fixed set of operators (say +, −, *, /, =) together with arbitrary variable names would match the same computer program with variable names changed.

The adaptability of the prefix array suggests looking at algorithms on regular strings that make use of the border array, with a view to (1) using instead the prefix array in order to improve performance; (2) extending the scope to indeterminate strings.

In the next two sections we survey recent work that implements this strategy.

3. Approximate Matching Under Hamming Distance

In DNA sequences it often happens that segments of the genome (substrings) are copied from one area to another, but with transcription errors, such as a change from *a* to *t* (substitution) or deletion/insertion of one or more letters. For this reason it becomes important to do approximate string matching that is tolerant of these errors. In this section, drawing on [37], we outline applications of the prefix array to this problem in the case that the errors are substitutions.

The ***Hamming distance*** $H(\boldsymbol{x}, \boldsymbol{w})$ between strings $\boldsymbol{x}$ and $\boldsymbol{w}$, both of length n, both perhaps containing indeterminate letters, is the number of positions $i \in 1..n$ such that $\boldsymbol{x}[i] \approx \boldsymbol{w}[i]$. Given a nonnegative integer $k < n$, we write $\boldsymbol{x} \equiv \pi_k^H \boldsymbol{w}$ if $H(\boldsymbol{x}, \boldsymbol{w}) \leq k$. Then the ***k-prefix table*** $\boldsymbol{\pi_k^H}$ of $\boldsymbol{x}$ under Hamming distance can be defined as follows: for every $i \in 1..n$, $\pi_k^H[i] = u$ is the length of the longest prefix of $\boldsymbol{x}$ such that $\boldsymbol{x}[i..i + u - 1] \equiv {}_k^H\boldsymbol{x}[1..u]$.

In [37] a very simple algorithm *kHP* is described that for regular strings computes π_k^H in expected time $O(kn)$ using only constant additional space. *kHP* is very fast in practice, based on the observation that the expected number of letter comparisons required at each position is less than 3. The algorithm also executes very well on indeterminate strings, though with an unspecified time bound. In addition the authors describe a much more complex algorithm, *kHP**, that uses global data structures and techniques to compute π_k^H in *guaranteed* $O(kn)$ worst case time; however, *kHP** is much slower in practice than *kHP* and does not extend to indeterminate strings. *kHP* (respectively, *kHP**) is then used in two applications important in bioinformatics:

(1) Given a *pattern* $\boldsymbol{u} = \boldsymbol{u}[1..m]$, a text $\boldsymbol{x} = \boldsymbol{x}[1..n]$, and an integer threshold $k < m$, algorithm *kHPT* (respectively, *kHPT**) finds all the substrings $\boldsymbol{u}'$ in $\boldsymbol{x}$ such that $H(\boldsymbol{u}, \boldsymbol{u}') \leq k$. The first step of the algorithm forms $\boldsymbol{z} = \boldsymbol{xu}$ and then computes the regular prefix table π_0^H of $\boldsymbol{z}$. According to the tests conducted in [37], *kHPT* is an order of magnitude faster in practice than all other known algorithms, including that proposed in [21].

(2) Given a set of r strings and an error rate ε, algorithm *kHOverlap* finds, over all pairs of strings, their suffix/prefix matches (approximate overlaps) of maximum length u that are within Hamming distance $k = \lceil \varepsilon u \rceil$. This procedure depends upon the efficient conversion of π_k^H to a

k-approximate border array β_k^H.

Also in [37] the edit distance $E(\boldsymbol{x}, \boldsymbol{w})$ between two strings is considered: the minimum number of edit operations (insert, delete, substitute) required to transform $\boldsymbol{x}$ into $\boldsymbol{w}$, where now we write $\boldsymbol{x} \approx_k^E \boldsymbol{w}$ if $E(\boldsymbol{x}, \boldsymbol{w}) < k$. Then the ***k-prefix table π_k^E under edit* distance** is analogously defined. An efficient algorithm *kEP* is described, using the dynamic programming matrix for $\boldsymbol{x}$ and $\boldsymbol{w}$, to compute π_k^E, but no algorithms *kEPT* and *kEOverlap*, analogous to *kHPT* and *kHOverlap*, are proposed.

Of course Hamming distance is of interest in computational biology, but efficient algorithms using edit distance are of much greater importance: this is due to the frequent occurrence of insertion and deletion errors in the transcription of one section of the genome to another that make Hamming distance unhelpful. For example, $H(\boldsymbol{x}, \boldsymbol{w}) = 5$ for $\boldsymbol{x} = acgta$, $\boldsymbol{w} = cgtag$, while $E(\boldsymbol{x}, \boldsymbol{w}) = 2$ (delete $\boldsymbol{x}[1] = a$, insert $\boldsymbol{x}[5] = g$). Thus, as discussed in Section 5, the further application of π_k^E is a priority.

4. Computing Covers of Strings

As noted in Observation 1, the borders (equivalently, the periods) of a string can provide an economical way of describing a string in certain cases. For example, the string $\boldsymbol{x} = cgatcgatcg$ shown in Figure 1 could be described by the triple $(p, e, t) = (4, 2, 2)$, telling us that $\boldsymbol{x}$ has period $p = 4$, ***exponent*** $e = 2$, and ***tail*** of length $t = 2$; that is, $\boldsymbol{x} = (cgat)^2(cg)$. Unfortunately, most strings cannot be described so simply; for example, $\boldsymbol{x'} = cagtcgatcg$, with just two letters interchanged, has only the empty border.

Thus it is of interest to identify other patterns in strings that might more often provide a basis for a short space-saving description. This requirement is of particular interest when the strings are both many and very long, as occurs in DNA sequence analysis.

In the early 1990s Apostolico and his co-authors [38,39] took a first step by introducing the cover of a string $\boldsymbol{x}$; that is, a nonempty proper border $\boldsymbol{u}$ of $\boldsymbol{x}$ such that every position i in $\boldsymbol{x}$ falls within an occurrence of $\boldsymbol{u}$. They called $|\boldsymbol{u}|$ the ***quasiperiod*** of $\boldsymbol{x}$. For example, $\boldsymbol{u} = cgatcg$ is a cover of $\boldsymbol{x} = cgatcgatcg$, which therefore has ***quasiperiod*** 6. Several papers were written to compute the covers of a given string $\boldsymbol{x}[1..n]$ [40–42], culminating in an algorithm [43] that, using the border array as a starting point, in linear time computed the cover array; that is, an array $\gamma[1..n]$ that, analogous to the border array, specifies all the quasiperiods of every prefix of $\boldsymbol{x}$.

Several efforts have been made to identify a more useful notion of cover. In [44] the ***seed*** of a string $\boldsymbol{x}$ was introduced — that is, the cover of a superstring of $\boldsymbol{x}$, thus not constrained to be a border — then extended in [45] to an approximate seed. Ref. [46] introduced the idea of a k-cover of $\boldsymbol{x}$ — that is, a minimum cardinality set of substrings of length k that together cover $\boldsymbol{x}$ —, later shown to be NP-hard to compute [47].

Some years later it was shown that a "relaxed" k-cover could be approximated in polynomial time [48]. Most recently a variant of the suffix tree, a highly space-consuming data structure, was used [49,50] to compute α-partial covers and α-partial seeds of $\boldsymbol{x}$, where α is a preselected parameter

that specifies the minimum number of positions in $\boldsymbol{x}$ to be covered. Collectively, these methods, while interesting, were often expensive in their space and/or time requirements and did not provide a sufficiently compact representation for most strings.

In [51] the ***minimum enhanced cover*** of $\boldsymbol{x}$ ($MEC_{\boldsymbol{x}}$) was proposed — that is, a border of $\boldsymbol{x}$ that is the shortest among all the borders that cover a maximum number of positions in $\boldsymbol{x}$. Also proposed were relaxed variants of $MEC_{\boldsymbol{x}}$ that did not require the cover to be a border of $\boldsymbol{x}$, but only a proper prefix. However, like the cover array algorithm, these methods depended strongly on an initial border array computation. Since, as we have seen, the prefix array has a flexibility that the border array does not, the possibility arises that these calculations can be handled using the prefix array instead, thus opening the way for the extension of covering algorithms to indeterminate strings.

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

Definition 1. *A string $x = x[1..n]$ 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,

$$\boldsymbol{x} = \{a, g\}\ c\ \{a, c\}\ \{a, c\}\ ca \quad (1)$$

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

However, note that the very concept of "regularity of a string" in some sense breaks down when we consider a sliding cover: now the "cover" need not actually "match" the area it is covering. In fact, a string can be a sliding cover of an indeterminate string $\boldsymbol{x}$ without being a substring of $\boldsymbol{x}$ 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 x of length κ. A rooted cover is defined simply by changing "suffix" to "prefix" in part (b) of Definition 1. The example string (1) has no rooted cover, but the string $\boldsymbol{x} = \{a, g\}\ c\ \{a, c\}\ \{a, c\}\ ac$ has both a sliding cover and a rooted cover of length 2. Note that, while a rooted cover must occur as a prefix of $\boldsymbol{x}$, it need not occur elsewhere in the string.

The first computation of the covers of an indeterminate string $\boldsymbol{x}$, in [52], begins by computing the "deterministic border array" of $\boldsymbol{x}$ as proposed in [22], then applies the Aho-Corasick automaton to determine whether each border can be a (rooted) cover [53]. Their algorithm requires $O(n)$ time on average to compute the largest cover of $\boldsymbol{x}$ itself; it is then iterated to compute the rooted cover array of $\boldsymbol{x}$ in worst case time $O(n^2)$.

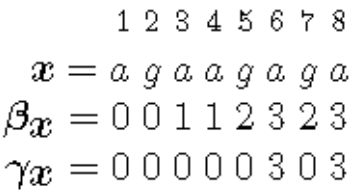

Figure 3. The border and cover arrays of $x = agaagaga$: even though the border array has many non-zero entries, the cover array is sparse.

Ref. [54] avoids any border computation altogether. For regular strings $\boldsymbol{x}$ the prefix array is computed in $O(n)$ worst-case time, then applied directly to compute the cover array, also in $O(n)$ time, and significantly faster in practice than the border-based method proposed in [43]. For indeterminate strings $\boldsymbol{x}$ the same general approach is followed to compute the rooted cover array, still requiring $O(n^2)$ time in the worst case, but now only $O(n)$ time on average.

It is encouraging that the prefix array can be applied to efficient computation of the rooted cover array for indeterminate strings, but still, as we have seen, the cover array has limited utility as an economical means of describing the patterns in a string. Perhaps more interesting is the use of the prefix table to extend the computation of the minimum enhanced cover $MEC_{\boldsymbol{x}}$ to indeterminate strings, as proposed in [55]. In fact, this paper shows how the prefix array can be used, instead of the border array, to compute $MEC_{\boldsymbol{x}}$ and its variants with identical worst-case asymptotic complexity but with much lower space requirements and, according to tests, significantly faster in practice. Then, of course, the prefix array is employed to extend these results in a straightforward fashion, based on the rooted cover, to indeterminate strings. It turns out, surprisingly, that, for indeterminate strings as for regular strings, $MEC_{\boldsymbol{x}}$ and its variants can be computed in $O(n)$ expected time.

Overall we have seen in this section that the prefix array can provide significant benefits, both in terms of efficiency and extended applicability, to methods that seek compressed representations of strings.

5. Open Problems

Even though, as we have seen, the prefix array has been used for more to be explored. Here we briefly note a few possible future directions of research:

i. Due to the importance of edit distance between strings in biological applications, it would seem to be of great interest to investigate algorithms that, given string $\boldsymbol{u}$ and integer $k > 0$, use the prefix array to find all substrings $\boldsymbol{u}$' in $\boldsymbol{x}$ such that the edit distance

$E(\boldsymbol{u}, \boldsymbol{u}') \leq k$. The results for Hamming distance using π_k^H look very promising, but nothing is known about the extension to edit distance and π_k^E .

ii. It turned out that the k-cover approach to compact representation of a string required an NP-hard computation, but are there other approaches to this problem that use multiple substrings to represent the given string $\boldsymbol{x}$ without this computational drawback?

iii. As we have seen, for regular strings the border array and the prefix array are equivalent in the sense that one can be computed from the other in linear time. Moreover, in at least some applications, making use of the prefix array rather than the border array provides benefit both in terms of algorithmic efficiency for regular strings and extendibility to indeterminate strings. What other such applications are there?

iv. More generally, in view of the prefix graph introduced in [33], what tools of graph theory can be brought to bear on problems arising in sequence analysis?

Notes:

[1] Technical terminology relevant to this survey is defined in Section 2; for other terms mentioned here, see for example [18,26,27].

[2] ***Nontandem repeats***, also of great interest in bioinformatics applications, take the form $\boldsymbol{uvu}$ for some nonempty $\boldsymbol{v} \neq \boldsymbol{u}$. So far the prefix array has not found application in the computation of nontandem repeats.

[3] By $i \in i_1..i_2$, all variables integers, we mean $i_1 \leq i \leq i_2$.

[4] That is, finding all matches for a given (regular or indeterminate) string in another given (regular or indeterminate) string $\boldsymbol{x}$, $|\boldsymbol{u}| < |\boldsymbol{x}|$.

Acknowledgements

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

References

1. Iliopoulos CS, Mouchard L (1999) Quasiperiodicity and string covering. *Theoret Comput Sci* 218: 205-216.
2. Smyth WF (2013) Computing regularities in strings: a survey. *Europ J Combinatorics* 34: 3-14.
3. Aho AV, Hopcroft JE (1974) The design and analysis of computer algorithms. Pearson Education India.
4. Knuth DE, Morris, Jr JH, Pratt VR (1977) Fast pattern matching in strings. *SIAM J Computing* 6:

323-350.
5. Fredkin E (1960) Trie memory. *Commun Assoc Comput Mach* 3: 490-499.
6. Weiner P (1973) Linear pattern matching algorithms. In: Switching and Automata Theory, 1973, SWAT'08. IEEE Conference Record of 14th Annual Symposium on. IEEE: 1-11.
7. McCreight EM (1976) A space-economical suffix tree construction algorithm. *JACM* 23: 262-272.
8. Farach M (1997) Optimal su x tree construction with large alphabets. In: Proceedings of the 38th Annual Symposium on the Foundations of Computer Science, FOCS: 97.
9. Apostolico A (1985) The myriad virtues of subword trees. In: Combinatorial algorithms on words. Springer Berlin Heidelberg: 85-96.
10. Manber U, Myers GW (1990) Suffix arrays: a new method for on-line string searches, *Proc. First Annual ACM-SIAM Symp. Discrete Algs:* 319-327.
11. Manber U, Myers G (1993) Suffix arrays: a new method for on-line string searches. *SIAM J Comput* 22: 935-948.
12. Kärkkäinen J, Sanders P (2003) Simple linear work suffix array construction. In: International Colloquium on Automata, Languages, and Programming. Springer Berlin Heidelberg: 943-955.
13. Ko P, Aluru S (2003) Space efficient linear time construction of suffix arrays. In: Annual Symposium on Combinatorial Pattern Matching. Springer Berlin Heidelberg: 200-210.
14. Kim DK, Sim JS, Park H, et al. (2003) Linear-time construction of suffix arrays. In: Annual Symposium on Combinatorial Pattern Matching. Springer Berlin Heidelberg: 186-199.
15. Puglisi SJ, Smyth WF, Turpin AH (2007) A taxonomy of suffix array construction algorithms. *acm Computing Surveys (CSUR)* 39: 4.
16. Nong G, Zhang S, Chan WH (2009) Linear time suffix array construction using D-critical substrings. In: Annual Symposium on Combinatorial Pattern Matching. Springer Berlin Heidelberg: 54-67.
17. Abouelhoda MI, Kurtz S, Ohlebusch E (2004) Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms* 2: 53-86.
18. Smyth B (2003) *Computing patterns in strings*. Pearson Education.
19. Fischer MJ, Paterson MS (1974) *String-matching and other products* (No. MAC-TM-41). Massachusetts Inst. of Technology.
20. Blanchet-Sadri F (2007) Algorithmic combinatorics on partial words. CRC Press.
21. Abrahamson K (1987) Generalized string matching. *SIAM J Comput 16*:1039-1051.
22. Holub J, Smyth WF (2003) Algorithms on indeterminate strings. 36-45.
23. Holub J, Smyth W F, Wang S (2008) Fast pattern-matching on indeterminate strings. *J Discrete Algorithms* 6: 37-50.
24. Smyth WF, Wang S (2009) A new approach to the periodicity lemma on strings with holes. *Theoret Comput Sci* 410: 4295-4302.
25. Main MG, Lorentz RJ (1984) An O (n log n) algorithm for finding all repetitions in a string. *J*

Algorithms 5: 422-432.
26. Crochemore M, Hancart C, Lecroq T (2001) Algorithmique du texte. Paris: Vuiber: 347.
27. Crochemore M, Hancart C, Lecroq T (2007) Algorithms on strings. Cambridge University Press: 383.
28. Smyth WF, Wang S (2008) New perspectives on the prefix array. In: International Symposium on String Processing and Information Retrieval. Springer Berlin Heidelberg: 133-143.
29. Bland W, Kucherov G, Smyth WF (2013) Prefix table construction and conversion. In: International Workshop on Combinatorial Algorithms. Springer Berlin Heidelberg: 41-53.
30. Gusfield D (1997) Algorithms on strings, trees and sequences: computer science and computational biology. Cambridge university press.
31. Clément J, Crochemore M, Rindone G (2009) Reverse engineering prefix tables. In: 26th International Symposium on Theoretical Aspects of Computer Science STACS 2009. IBFI Schloss Dagstuhl: 289-300.
32. Alatabbi A, Rahman MS, Smyth WF (2015) Inferring an indeterminate string from a prefix graph. *J Discrete Algorithms* 32: 6-13.
33. Christodoulakis M, Ryan PJ, Smyth WF, et al. (2015) Indeterminate strings, prefix arrays & undirected graphs. *Theoret Comput Sci* 600: 34-48.
34. Blanchet-Sadri F, Bodnar M, De Winkle B (2017) New bounds and extended relations between prefix arrays, border arrays, undirected graphs, and indeterminate strings. *Theory of Computing Systems* 60: 473-497.
35. Helling J, Ryan PJ, Smyth WF, et al. (2017) Constructing an indeterminate string from its associated graph. *Theoret Comput Sci.* Available from: http://www.sciencedirect.com/science/article/pii/S0304397517301494
36. Beal R, Adjeroh DA, Smyth WF (2017) A prefix array for parameterized strings. *J Discrete Algorithms* 42: 23-34.
37. Barton C, Iliopoulos CS, Pissis SP, et al. (2014) Fast and simple computations using prefix tables under hamming and edit distance. In: International Workshop on Combinatorial Algorithms. Springer International Publishing: 49-61.
38. Apostolico A, Ehrenfeucht A (1993) Efficient detection of quasiperiodicities in strings. *Theoret Comput Sci* 119: 247-265.
39. Apostolico A, Farach M, Iliopoulos CS (1991) Optimal superprimitivity testing for strings. *Inform Process Lett* 39: 17-20.
40. Breslauer D (1992) An on-line string superprimitivity test. *Inform Proces. Lett* 44: 345-347.
41. Moore D, Smyth WF (1994) An optimal algorithm to compute all the covers of a string. *Inform Process Lett* 50: 239-246.
42. Moore D, Smyth WF (1995) A correction to "An optimal algorithm to compute all the covers of a string". *Inform Process Lett* 54: 101-103.
43. Li Y, Smyth WF (2002) Computing the cover array in linear time. *Algorithmica* 32: 95-106.

44. Iliopoulos CS, Moore DWG, Park K (1996) Covering a string. *Algorithmica* 16: 288-297.
45. Christodoulakis M, Iliopoulos CS, Park K, et al. (2005) Approximate Seeds of Strings. *J Automata Languages & Combinatorics* 10: 609-626.
46. Iliopoulos CS, Smyth WF (1998) On-line algorithms for k-covering. Proc. 9th Australasian Workshop on Combinatorial Algs: 107-116.
47. Cole R, Ilopoulos CS, Mohamed M, et al. (2005) The complexity of the minimum k-cover problem. *J Automata Languages & Combinatorics* 10: 641-653.
48. Iliopoulos CS, Mohamed M, Smyth WF (2011) New complexity results for the k-covers problem. *Inform Sci* 181: 2571-2575.
49. Kociumaka T, Pissis SP, Radoszewski J, et al. (2015) Fast algorithm for partial covers in words. *Algorithmica* 73: 217-233.
50. Kociumaka T, Pissis S P, Radoszewski J, et al. (2016) Efficient algorithms for shortest partial seeds in words. *Theoret Comput Sci* Available from: http://www.sciencedirect.com/science/article/pii/S0304397516307034
51. Flouri T, Iliopoulos CS, Kociumaka T, et al. (2013) Enhanced string covering. *Theoret Comput Sci* 506: 102-114.
52. Bari MF, Rahman MS, Shahriyar R (2009) Finding All Covers of an Indeterminate String in O (n) Time on Average. In: Stringology: 263-271.
53. Aho AV, Corasick MJ (1975) Efficient string matching: an aid to bibliographic search. *Commun Assoc Comput Mach* 18: 333-340.
54. Alatabbi A, Rahman MS, Smyth WF (2016) Computing covers using prefix tables. *Discrete Appl Math* 212: 2-9.
55. Alatabbi A, Islam AS, Rahman MS, et al. (2016) Enhanced covers of regular & indeterminate strings using prefix tables. *J Automata Languages & Combinatorics*: 41-46.

AIMS Press