Inferring an indeterminate string from a prefix graph

Ali Alatabbi, M. Sohel Rahman, W.F. Smyth

PII: S1570-8667(14)00099-9
DOI: 10.1016/j.jda.2014.12.006
Reference: JDA 593

To appear in: *Journal of Discrete Algorithms*

Received date: 19 January 2014
Revised date: 12 December 2014
Accepted date: 18 December 2014

Please cite this article in press as: A. Alatabbi et al., Inferring an indeterminate string from a prefix graph, *Journal of Discrete Algorithms* (2014), http://dx.doi.org/10.1016/j.jda.2014.12.006

Inferring an Indeterminate String from a Prefix Graph

Ali Alatabbi[1], M. Sohel Rahman*[2], and W. F. Smyth**[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
[4] School of Engineering & Information Technology,
Murdoch University
smyth@mcmaster.ca

Abstract. An ***indeterminate string*** (or, more simply, just a ***string***) $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on an alphabet Σ is a sequence of nonempty subsets of Σ. We say that $\boldsymbol{x}[i_1]$ and $\boldsymbol{x}[i_2]$ ***match*** (written $\boldsymbol{x}[i_1] \approx \boldsymbol{x}[i_2]$) if and only if $\boldsymbol{x}[i_1] \cap \boldsymbol{x}[i_2] \neq \emptyset$. A ***feasible array*** is an array $\boldsymbol{y} = \boldsymbol{y}[1..n]$ of integers such that $\boldsymbol{y}[1] = n$ and for every $i \in 2..n$, $\boldsymbol{y}[i] \in 0..n-i+1$. A ***prefix table*** of a string $\boldsymbol{x}$ is an array $\boldsymbol{\pi} = \boldsymbol{\pi}[1..n]$ of integers such that, for every $i \in 1..n$, $\boldsymbol{\pi}[i] = j$ if and only if $\boldsymbol{x}[i..i+j-1]$ is the longest substring at position i of $\boldsymbol{x}$ that matches a prefix of $\boldsymbol{x}$. It is known from [6] that every feasible array is a prefix table of some indeterminate string. A ***prefix graph*** $\mathcal{P} = \mathcal{P}_{\boldsymbol{y}}$ is a labelled simple graph whose structure is determined by a feasible array $\boldsymbol{y}$. In this paper we show, given a feasible array $\boldsymbol{y}$, how to use $\mathcal{P}_{\boldsymbol{y}}$ to construct a lexicographically least indeterminate string on a minimum alphabet whose prefix table $\boldsymbol{\pi} = \boldsymbol{y}$.

1 Introduction

In the extensive literature of stringology/combinatorics on words, a "string" or "word" has usually been defined as a sequence of individual elements of a distinguished set Σ called an "alphabet". Nevertheless, going back as far as the groundbreaking paper of Fischer & Paterson [9], more general sequences, defined instead on *subsets* of Σ, have also been considered. The more constrained model introduced in [9] restricts entries in a string to be either elements of Σ (subsets of size 1) or Σ itself (subsets of size $\sigma = |\Sigma|$); these have been studied in recent years as "strings

* Supported by an ACU Titular Fellowship.
** Supported in part by the Natural Sciences & Engineering Research Council of Canada.

with don't cares" [14], also "strings with holes" or "partial words" [3]. The unconstrained model, which allows arbitrary nonempty subsets of Σ, has also attracted significant attention, often because of applications in bioinformatics: such strings have variously been called "generalized" [1], "indeterminate" [13], or "degenerate" [15].

In this paper we study strings in their full generality, hence the following definitions:

Definition 1. *Suppose a set Σ of symbols (called the **alphabet**) is given. A **string** $\boldsymbol{x}$ on Σ of **length** $n = |\boldsymbol{x}|$ is a sequence of $n \geq 0$ nonempty finite subsets of Σ, called **letters**; we represent $\boldsymbol{x}$ as an array $\boldsymbol{x}[1..n]$. If $n = 0$, $\boldsymbol{x}$ is called the **empty string** and denoted by $\boldsymbol{\varepsilon}$; if for every $i \in 1..n$, $\boldsymbol{x}[i]$ is a subset of Σ of size 1, $\boldsymbol{x}$ is said to be a **regular string**.*

Definition 2. *Suppose we are given two strings $\boldsymbol{x}$ and $\boldsymbol{y}$ and integers $i \in 1..|\boldsymbol{x}|$, $j \in 1..|\boldsymbol{y}|$. We say that $\boldsymbol{x}[i]$ and $\boldsymbol{y}[j]$ **match** (written $\boldsymbol{x}[i] \approx \boldsymbol{y}[j]$) if and only if $\boldsymbol{x}[i] \cap \boldsymbol{y}[j] \neq \emptyset$. Then $\boldsymbol{x}$ and $\boldsymbol{y}$ **match** ($\boldsymbol{x} \approx \boldsymbol{y}$) if and only if $|\boldsymbol{x}| = |\boldsymbol{y}|$ and $\boldsymbol{x}[i] \approx \boldsymbol{y}[i]$ for every $i \in 1..|\boldsymbol{x}|$.*

Note that matching is not necessarily transitive: $a \approx \{a, b\} \approx b$, but $a \not\approx b$.

Definition 3. *The **prefix table** (also **prefix array**)[5] of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is the integer array $\boldsymbol{\pi_x} = \boldsymbol{\pi_x}[1..n]$ such that for every $i \in 1..n$, $\boldsymbol{\pi_x}[i]$ is the length of the longest prefix of $\boldsymbol{x}[i..n]$ that matches a prefix of $\boldsymbol{x}$. Thus for every prefix table $\boldsymbol{\pi_x}$, $\boldsymbol{\pi_x}[1] = n$. When there is no ambiguity, we write $\boldsymbol{\pi} = \boldsymbol{\pi_x}$.*

The prefix table is an important data structure for strings: it identifies all the borders, hence all the periods, of every prefix of $\boldsymbol{x}$ [6]. It was originally introduced to facilitate the computation of repetitions in regular strings [16], see also [20]; and for regular strings, prefix table and border array are equivalent, since each can be computed from the other in linear time [5]. For general strings, the prefix table can be computed in compressed form in $O(n^2)$ time using $\Theta(n/\sigma)$ bytes of storage space [21], where $\sigma = |\Sigma|$. Two examples follow, adapted from [6]:

$$\begin{array}{r} \phantom{\boldsymbol{x_1}} \; 1\;2\;3\;4\;5\;6\;7\;8 \\ \boldsymbol{x_1} = a\;c\;a\;g\;a\;c\;a\;t \\ \boldsymbol{\pi_1} = 8\;0\;1\;0\;3\;0\;1\;0 \end{array} \tag{1}$$

[5] We prefer "table" because of the possible confusion with "suffix array", a completely different data structure.

$$\begin{array}{rcccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \boldsymbol{x_2} = & \{a,c\} & \{g,t\} & \{a,g\} & \{a,c,g\} & g & c & \{a,t\} & a \\ \boldsymbol{\pi_2} = & 8 & 0 & 4 & 2 & 0 & 3 & 1 & 1 \end{array} \tag{2}$$

Since clearly every position $i \in 2..n$ in a prefix table $\boldsymbol{\pi}$ must satisfy $0 \leq \boldsymbol{\pi}[i] \leq n-i+1$, the following definition is a natural one:

An array $\boldsymbol{y} = \boldsymbol{y}[1..n]$ of integers is said to be a ***feasible array*** if and only if $\boldsymbol{y}[1] = n$ and for every $i \in 2..n$, $\boldsymbol{y}[i] \in 0..n-i+1$.

An immediate consequence of Definition 3 is the following:

Lemma 1 ([6]). *Let $\boldsymbol{x} = \boldsymbol{x}[1..n]$ be a string. An integer array $\boldsymbol{y} = \boldsymbol{y}[1..n]$ is the prefix table of $\boldsymbol{x}$ if and only if for each position $i \in 1..n$, the following two conditions hold:*

(a) $\boldsymbol{x}\big[1..\boldsymbol{y}[i]\big] \approx \boldsymbol{x}\big[i..i+\boldsymbol{y}[i]-1\big]$;
(b) if $i+\boldsymbol{y}[i] \leq n$, *then* $\boldsymbol{x}\big[y[i]+1\big] \not\approx \boldsymbol{x}\big[i+\boldsymbol{y}[i]\big]$.

Then the following fundamental result establishes the important connection between strings and feasible arrays:

Lemma 2 ([6]). *Every feasible array is the prefix table of some string.*

In view of this lemma, we say that a feasible array is ***regular*** if it is the prefix array of a regular string. We are now able to state the goal of this paper as follows: for a given feasible array $\boldsymbol{y} = \boldsymbol{y}[1..n]$, not necessarily regular, construct a string $\boldsymbol{x}$ on a minimum alphabet whose prefix table $\boldsymbol{\pi_x} = \boldsymbol{y}$ — the "reverse engineering" problem for the prefix table in its full generality. In fact, we do somewhat more: we construct a lexicographically least such string, in a sense to be defined in Section 2.

The first reverse engineering problem in stringology was stated and solved in [11, 10], where an algorithm was described to compute a lexicographically least regular string whose border array was a given integer array — or to return the result that no such regular string exists. Many other similar constructions have since been published, related to other stringological data structures but always specific to regular strings (see [2, 8, 12, 18], among others). [19] was the first paper to consider the more general problem of inferring an indeterminate string from a given data structure (specifically, border array, suffix array and LCP array). Although solving such problems does not yield immediate applications, nevertheless solutions provide a deeper understanding of the combinatorial many-many relationship between strings and the various data structures developed from them (see for example [17], where canonical strings corresponding

to given border arrays are identified and efficiently generated for use as test data).

For prefix tables and regular strings, the reverse engineering problem was solved in [7], where a linear-time algorithm was described to return a lexicographically least regular string $\boldsymbol{x}$ whose prefix table is the given feasible array $\boldsymbol{y}$, or an error message if no such $\boldsymbol{x}$ exists. A recent paper [4] sketches two algorithms to compute an indeterminate string $\boldsymbol{x}$ on a minimum alphabet (not necessarily lexicographically least) corresponding to a given feasible array $\boldsymbol{y}$, but the algorithms are theoretical in nature: one requires the determination of the chromatic number of a certain graph, an NP-hard problem, while the other depends on somehow identifying the minimum "induced positive edge cover" of a graph. However, [4] proves an important result that we use below to bound the complexity of our algorithm: that the minimum alphabet size σ of a string corresponding to a given feasible array of length n is at most $n+\sqrt{n}$. In this paper we use graph-theoretic methods developed from [6] to compute a lexicographically least string, regular or not, corresponding to the given $\boldsymbol{y}$, in time $O(\sigma n^2)$.

Section 2 of this paper provides background material for an understanding of our algorithm; Section 3 presents the algorithm itself; Section 4 briefly discusses these results and suggests future work.

2 Preliminaries

Following [6], for a given feasible array $\boldsymbol{y} = \boldsymbol{y}[1..n]$, we define a corresponding graph $\mathcal{P} = \mathcal{P}_{\boldsymbol{y}}$, on which our algorithm will be based:

Definition 4. *Let $\mathcal{P} = (V, E)$ be a labelled graph with vertex set $V = \{1, 2, \ldots, n\}$ consisting of positions in a given feasible array $\boldsymbol{y}$. In $\mathcal{P}$ we define, for $i \in 2..n$, two kinds of edge (compare Lemma 1):*

*(a) for every $h \in 1..\boldsymbol{y}[i]$, $(h, i+h-1)$ is called a **positive edge**;*
*(b) $(1+\boldsymbol{y}[i], i+\boldsymbol{y}[i])$ is called a **negative edge**, provided $i+\boldsymbol{y}[i] \le n$.*

*E^+ and E^- denote the sets of positive and negative edges, respectively. We write $E = E^+ \cup E^-$, $\mathcal{P}^+ = (V, E^+)$, $\mathcal{P}^- = (V, E^-)$, and we call $\mathcal{P}$ the **prefix graph $\mathcal{P}_{\boldsymbol{y}}$** of $\boldsymbol{y}$. If $\boldsymbol{x}$ is a string having $\boldsymbol{y}$ as its prefix table, then we also refer to $\mathcal{P}$ as the **prefix graph $\mathcal{P}_{\boldsymbol{x}}$** of $\boldsymbol{x}$.*

Observe that E^+ and E^- are necessarily disjoint. Figures 1–4 show the prefix graphs for the example strings (3) and (4).

$$\begin{array}{rcccccccccc} & & 1 & & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \boldsymbol{x_3} = & \{a,b\} & \{a,c\} & c & \{a,b\} & b & c & \{a,c\} & b \\ \boldsymbol{\pi_3} = & & 8 & & 2 & 0 & 1 & 4 & 0 & 1 & 1 \end{array} \tag{3}$$

$$\begin{array}{rcccccccccc} & & 1 & & 2 & & 3 & & 4 & 5 & 6 & 7 & 8 \\ \boldsymbol{x_4} = & \{a,b\} & & \{a,c\} & & \{a,d\} & & \{c,e\} & a & \{b,e\} & c & d \\ \boldsymbol{\pi_4} = & & 8 & & 2 & & 4 & & 0 & 1 & 3 & 0 & 0 \end{array} \tag{4}$$

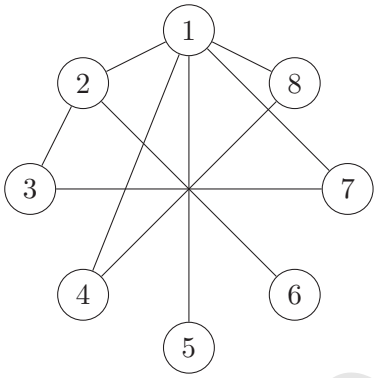

Fig. 1. $\mathcal{P}^{+}_{\boldsymbol{y_3}}$ for $\boldsymbol{y_3} = 82014011$

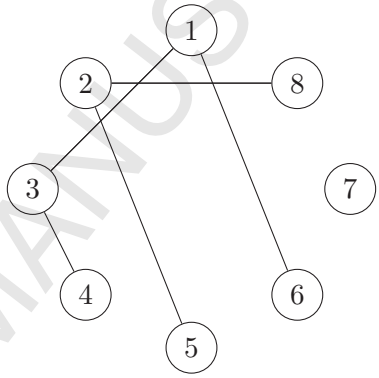

Fig. 2. $\mathcal{P}^{-}_{\boldsymbol{y_3}}$ for $\boldsymbol{y_3} = 82014011$

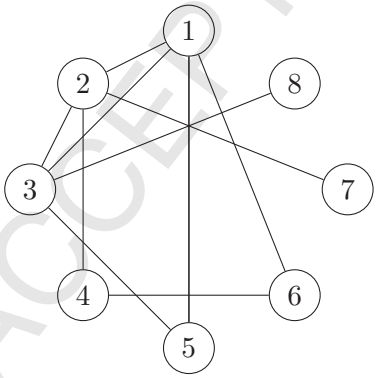

Fig. 3. $\mathcal{P}^{+}_{\boldsymbol{y_4}}$ for $\boldsymbol{y_4} = 82401300$

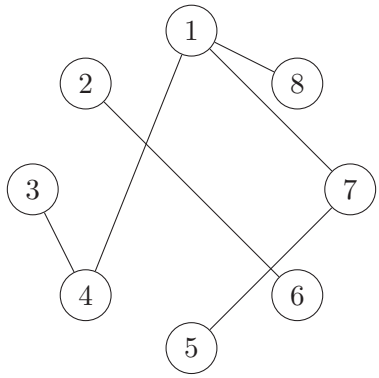

Fig. 4. $\mathcal{P}^{-}_{\boldsymbol{y_4}}$ for $\boldsymbol{y_4} = 82401300$

The following lemma will be useful for the analysis of our algorithm:

Lemma 3 ([6]). *Let $\mathcal{P}_{\boldsymbol{y}} = (V, E)$ be a prefix graph of a feasible array $\boldsymbol{y}$. Then $\boldsymbol{y}$ is regular if and only if every edge of $P^{-}_{\boldsymbol{y}}$ joins two vertices in distinct connected components of $P^{+}_{\boldsymbol{y}}$.*

So as to discuss the lexicographical ordering of strings on an ordered alphabet Σ, we need first of all a definition of the order of two letters:

Definition 5. *Suppose two letters λ and μ are given, where*

$$\lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_j\},\ \mu = \{\mu_1, \mu_2, \ldots, \mu_k\},$$

with $\lambda_h \in \Sigma$ for every $h \in 1..j$, $\mu_h \in \Sigma$ for every $h \in 1..k$. We assume without loss of generality that $j \leq k$, also that $\lambda_h < \lambda_{h+1}$ for every $h \in 1..j-1$ and $\mu_h < \mu_{h+1}$ for every $h \in 1..k-1$. Then $\lambda = \mu$ if and only if $\lambda_h = \mu_h$ for every $h \in 1..k$ and $j = k$; while $\lambda \prec \mu$ if and only if

(a) $\lambda_h = \mu_h$ for every $h \in 1..j < k$; or
(b) $\lambda_h = \mu_h$ for every $h \in 1..h' < j$ **and** *$\lambda_{h'+1} < \mu_{h'+1}$.*

Otherwise, $\mu \prec \lambda$.

Note that $(\lambda = \mu) \Rightarrow (\lambda \approx \mu)$, but that $\lambda \approx \mu$ implies neither equality nor an ordering of λ and μ. We remark also that the definition of letter order given here is not the only possible or useful one. For example, it would require $\{a, b, w, x, y, z\} \prec \{a, c\}$, thus arguably not placing sufficient emphasis on economy of letter selection in the alphabet.

Definition 6. *Now suppose that two strings $\boldsymbol{x_1} = \boldsymbol{x_1}[1..n_1]$ and $\boldsymbol{x_2} = \boldsymbol{x_2}[1..n_2]$ on Σ are given, where without loss of generality we assume that $n_1 \leq n_2$. Then $\boldsymbol{x_1} = \boldsymbol{x_2}$ if and only if $\boldsymbol{x_1}[h] = \boldsymbol{x_2}[h]$ for every $h \in 1..n_2$ and $n_1 = n_2$; while $\boldsymbol{x_1} \prec \boldsymbol{x_2}$ if and only if*

(a) $\boldsymbol{x_1}[h] = \boldsymbol{x_2}[h]$ for every $h \in 1..n_1 < n_2$; or
(b) $\boldsymbol{x_1}[h] = \boldsymbol{x_2}[h]$ for every $h \in 1..h' < n_1$ **and** *$\boldsymbol{x_1}[h'+1] \prec \boldsymbol{x_2}[h'+1]$.*

Otherwise, $\boldsymbol{x_2} \prec \boldsymbol{x_1}$.

To better illustrate the relation of strings defined and used in this paper we present the following examples:

- $\boldsymbol{x_1} = \{a, c\}\ \{g, t\}\ a \prec \boldsymbol{x_2} = \{a, c\}\ \{g, t\}\ \{a, g\}$
- $\boldsymbol{x_1} = a\ \{g, t\}\ \{a, c\}\ \{a, c, g\} \prec \boldsymbol{x_2} = a\ \{g, t\}\ \{a, t\}\ a$
- $\boldsymbol{x_1} = a\ \{a, c, g\}\ \{a, c, g\}\ \{a, t\} \prec \boldsymbol{x_2} = \ \{a, c\}\ g\ g\ \{a, t\}$

where $a < c < g < t$.

3 Algorithm RevEng

3.1 The Algorithm

The basic strategy of Algorithm RevEng, that constructs a lexicographically least string $\boldsymbol{x}$ (initially empty) corresponding to a given feasible array $\boldsymbol{y} = \boldsymbol{y}[1..n]$, is expressed by the main steps given below. Initially the alphabet Σ is empty ($\sigma = 0$), as are the sets $\boldsymbol{x}[i]$, $1 \leq i \leq n$.

(S1) Consider the edges (i, j) of E^+ in increasing order of $ni+j$ in order to add a single letter to $\boldsymbol{x}[i]$, $\boldsymbol{x}[j]$, or both based on the following steps;

(S2) if, by virtue of previous assignments, $\boldsymbol{x}[i] \approx \boldsymbol{x}[j]$ (so neither is empty), there is nothing to do — (i, j) can be skipped;

(S3) otherwise, for the current (i, j), determine a sequence

$$C = (\lambda_1, i_1), (\lambda_2, i_2), \ldots, (\lambda_r, i_r)$$

of all candidate assignments, where for every $h \in 1..r$, $i_h = i$ (respectively, j) if $\lambda_h \in \boldsymbol{x}[j]$ (respectively, $\boldsymbol{x}[i]$), and $\lambda_1 < \lambda_2 < \cdots < \lambda_r$;

(S4) for the current h, determine whether or not the assignment

$$\boldsymbol{x}[i_h] \leftarrow \boldsymbol{x}[i_h] \cup \{\lambda_h\}$$

is "allowable" (that is, compatible with the neighbourhood of i_h in E^-) — if so, then perform the assignment, maintaining the elements of $\boldsymbol{x}[i_h]$ in their natural order;

(S5) if for no h is the assignment allowable, then assign a least new letter (drawn WLOG from the set of positive integers) to both $\boldsymbol{x}[i]$ and $\boldsymbol{x}[j]$;

(S6) since it may be that after Steps (S1)-(S5) have been executed for every $(i, j) \in E^+$, there still remain unassigned positions in $\boldsymbol{x}$ (that is, corresponding to isolated vertices in $\mathcal{P}^+$), a final assignment of a least possible letter for these positions is required (see function $\texttt{LEAST}(\texttt{i}, \lambda_{\max})$ and Lemma 4).

In order to implement this algorithm, several data structures need to be created, maintained, and accessed:

(DS1) The edges of E^+ are made accessible in increasing order for Step (S1) by a radix sort of the positive edges (i, j) into a linked list L^+ (i.e., the linked list L^+ contains the edges of E^+ in increasing order), whose entries occur in increasing order of i and, within each i, in increasing order of j. The time requirement is $\Theta(|E^+|)$, thus $O(n^2)$ in the worst case, since E^+ can contain $\Theta(n^2)$ edges [6].

(DS2) In order to implement Step (S4) of Algorithm RevEng, we need, for each position $i \in 1..n$, to have available a linked list of positions j such that (i, j) is an edge of E^-. This can be done by using E^- to form a set of negative edges that includes each (i, j) twice, both as (i, j) and as (j, i). Then in a preprocessing step the entries in this set are radix sorted into an array $N^- = N^-[1..n]$ of n linked lists, such that for every $i \in 1..n$, $N^-[i]$ gives in increasing order all the vertices j for which $(i, j) \in E^-$ (in other words, the neighbourhood of i in E^-). Since E^- contains $O(n)$ edges [6], this preprocessing step can be accomplished in $O(n)$ time.

(DS3) Steps (S2)-(S5) require that for each $i \in 1..n$, a linked list $\boldsymbol{x}[i]$ be maintained of letters λ that have so far been assigned to $\boldsymbol{x}[i]$. Each list is maintained in increasing letter order, so that update, intersection, and union each require $O(\sigma)$ time, where $\sigma = |\Sigma|$ is the (current) alphabet size. Since for regular strings each $\boldsymbol{x}[i]$ has exactly one element, in this case processing time reduces to $O(1)$.

(DS4) In Step (S4), in order to determine whether a proposed assignment of a letter λ_h to a position i'_h in $\boldsymbol{x}$ is allowable or not, we form a "forbidden" matrix $F[1..n, 1..\sigma]$ in which $F[i, \lambda] = 1$ if and only if $\lambda \in \boldsymbol{x}[i]$ is forbidden. F is updated and used as follows:

- for each new letter $\lambda_{\max}$ introduced in Step (S5), $F[i, \lambda_{\max}]$ is initialized to zero for all $i \in 1..n$;
- whenever an assignment $\boldsymbol{x}[i] \stackrel{+}{\leftarrow} \lambda$ is made in Steps (S4) & (S5), set $F[j, \lambda] \leftarrow 1$ for every $j \in N^-[i]$ (procedure `UPDATE_F`(i, λ))`.`

Figures 5 and 6 give pseudocode for Algorithm RevEng and function LEAST, respectively.

3.2 Correctness

Consider first the main **while** loop of Algorithm RevEng, in which the edges of E^+ are considered in strict increasing (i, j) order. We see that new letters $\lambda_{\max}$ are first introduced at the leftmost possible positions in $\boldsymbol{x}$. Thereafter, whenever a letter is reused ($\lambda_{\max}$ not increased), it is always the minimum possible letter consistent with the least possible currently unfilled positions (i, j) that is used — by virtue of the fact that the entries in C are maintained in increasing order of λ. Thus any automorphism of the alphabet Σ other than the identity would yield a larger string. We conclude that within the main **while** loop the assignments maintain lexicographical order $\prec$ as defined in Section 2.

```
procedure RevEng (P, x, n)
λmax ← 0; x ← ∅^n; F[1..n, 1..σ] ← 0^{nσ}
while top(L+) ≠ ∅ do
    (i, j) ← pop(L+); C ← ∅          ▷ i < j; ni+j a minimum
    if x[i] ∩ x[j] = ∅ then
        ∀λ ∈ x[i] do C1 ←+ (λ, j)          ▷ ordered by λ
        ∀λ ∈ x[j] do C2 ←+ (λ, i)          ▷ ordered by λ
    ▷ Merge C1 and C2 into a single sequence ordered by λ.
        C ← MERGE(C1, C2)
        SET ← false
        while top(C) ≠ ∅ and not SET do
            (λ, h) ← pop(C)
            if F[h, λ] ≠ 1 then
                x[h] ←+ λ          ▷ maintain λ ordering
                SET ← true; UPDATE_F(h, λ)
        if not SET then
            λmax ← λmax+1
            for h ← 1 to n do F[h, λmax] ← 0
            x[i] ←+ λmax; UPDATE_F(i, λmax)
            x[j] ←+ λmax; UPDATE_F(j, λmax)
for i ← 1 to n do
    if x[i] = ∅ then
    ▷ Identify the least letter λ that does not occur
    ▷ in any x[j] for which j ∈ N−[i].
        λ ← LEAST(i, λmax); λmax ← max(λ, λmax)
        x[i] ← λ
```

Fig. 5. Given the preprocessing outlined in (DS1)-(DS2), Algorithm RevEng computes $\boldsymbol{x}[1..n]$, the lexicographically least string corresponding to a given prefix (feasible) graph $\mathcal{P}$ on n vertices.

It may happen, however, that certain positions i in $\boldsymbol{x}$ remain empty, those corresponding to isolated vertices i in $\mathcal{P}^+$. Assignments to these positions are handled by the final **for** loop, which we now consider.

Lemma 4. *A vertex $i \in 1..n$ is isolated in $\mathcal{P}^+$ if and only if*

(a) $\boldsymbol{y}[i] = 0$ *or* $i = 1$*;* **and**
(b) for every $j \in 2..n$, $\boldsymbol{y}[j] < i$*;* **and**
(c) for every $j \in 1..i-1$, $j+\boldsymbol{y}[j] \leq i$.

function LEAST(i, $\lambda_{\max}$)
$B[1..\lambda_{\max}] \leftarrow 0^{\lambda_{\max}}$
$\forall j \in N^-[i]$ **do**
    $\forall \lambda \in \boldsymbol{x}[j]$ **do**
        $B[\lambda] \leftarrow 1$
$\lambda \leftarrow 1$
while $\lambda \leq \lambda_{\max}$ **and** $B[\lambda] = 1$ **do**
    $\lambda \leftarrow \lambda{+}1$

Fig. 6. Identify the least letter λ that does **not** occur in **any** $\boldsymbol{x}[j]$ for which $j \in N^-[i]$.

Proof. First suppose that i is isolated. Then (a) must hold; otherwise, for $i > 1$ and $\boldsymbol{y}[i] > 0$, there exists an edge $(1, i) \in E^+$, a contradiction. If (b) does not hold, there exists $j > 1$ such that $\boldsymbol{y}[j] = r \geq i$, implying $\boldsymbol{x}[1..r] \approx \boldsymbol{x}[j..j{+}r{-}1]$, hence $\boldsymbol{x}[i] \approx \boldsymbol{x}[j{+}i{-}1]$, so that $(i, j{+}i{-}1) \in E^+$, again a contradiction. Similarly, if (c) does not hold, there exists $j \in 1..i{-}1$ such that $\boldsymbol{y}[j] = r$ with $j{+}r > i$. Consequently, $\boldsymbol{x}[j..j{+}r{-}1] \approx \boldsymbol{x}[1..r]$ implying $\boldsymbol{x}[i] \approx \boldsymbol{x}[i{-}j{+}1]$, so that $(i{-}j{+}1, i) \in E^+$, a contradiction that establishes sufficiency.

Suppose then that conditions (a)-(c) all hold. If we assume that $i = 1$ in (a), then (b) implies that for every $j \in 2..n$, $\boldsymbol{y}[j] = 0$, so that $E^+ = \emptyset$ and so every position i is isolated. Otherwise, if $\boldsymbol{y}[i] = 0$ for some $i > 1$, conditions (b) and (c) ensure that position i is not contained in any matching range within $\boldsymbol{x}$ and is therefore isolated in $\mathcal{P}^+$, as required. □

Since in the main **while** loop new maximum letters $\lambda_{\max}$ are introduced into pairs i and $j > i$ of positions in $\boldsymbol{x}$ that are determined by entries in $\boldsymbol{y}$, it is an immediate consequence of Lemma 4 that i must be less than any isolated vertex, in particular the smallest one, $i_{\min}$, say. In other words, every letter assigned during the execution of the main **while** loop occurs at least once to the left of $i_{\min}$ in $\boldsymbol{x}$. It follows that lexicographical order will be maintained if any required additional letters $\lambda_{\max}{+}1, \lambda_{\max}{+}2, \cdots$ are assigned to an ascending sequence of positions in $\boldsymbol{x}$ determined by the isolated vertices in $\mathcal{P}^+$. We have

Lemma 5. *Given a prefix graph $\mathcal{P}_{\boldsymbol{y}}$ corresponding to a feasible array $\boldsymbol{y}$, Algorithm RevEng constructs a lexicographically least indeterminate string on a minimum alphabet whose prefix table $\boldsymbol{\pi} = \boldsymbol{y}$.*

3.3 Asymptotic Complexity

The main **while** loop in Algorithm RevEng will be executed exactly $|E^+|$ times. Within the loop the construction of C requires time proportional to $|C| = |\boldsymbol{x}[i]| + |\boldsymbol{x}[j]|$, thus $O(\sigma)$ in the worst case. The processing of C then also requires $O(\sigma)$ time in the worst case, except for the time required for `UPDATE_F`. Each of the three calls of `UPDATE_F` corresponds to the assignment of a letter λ to a vertex i of $\mathcal{P}^-$ and the ensuing update of $F[i, \lambda]$, an event that can occur at most σ times for each edge in E^-, thus at most $(\sigma \times |E^-|)$ times overall. Similarly, the **for** loop that initializes the F array requires $\Theta(n)$ time for each of at most σ values of $\lambda_{\max}$.

We conclude that the worst-case time requirement for the **while** loop is $O\big(\sigma \max(|E^+|, |E^-|)\big)$. As illustrated by the examples $\boldsymbol{y} = n0^{n-1}$ and $\boldsymbol{y} = n|n-1|\ldots|1$, the bounds on these quantities are as follows: $0 \le |E^+| \le \binom{n}{2}$ and $0 \le |E^-| \le n-1$, while $|E^+|+|E^-| \ge n-1$. As noted earlier, it was shown in [4] that $\sigma \le n+\sqrt{n}$, with a further conjecture that $\sigma \le n$.

Turning our attention to the terminating **for** loop of Algorithm RevEng, we observe that for at most n executions of function `LEAST`, the binary array B of length at most σ must be created, thus overall consuming $O(\sigma n)$ time. The nested $\forall$ loops in `LEAST` set positions in B λ times for at most every edge in E^-, again requiring at most $O(\sigma n)$ time over all invocations of `LEAST`. Thus

Lemma 6. *Algorithm RevEng requires $O(\sigma n^2)$ time in the worst case, where $\sigma \le n+\sqrt{n}$.*

3.4 Example

Suppose $\boldsymbol{y} = 50210$, so that $E^+ = 13, 14, 24$ and $E^- = 12, 15, 25, 35$.

- In E^+ first consider edge 13, leading to assignments $\boldsymbol{x}[1] \leftarrow a$, $\boldsymbol{x}[3] \leftarrow a$, with $F[2, a] = F[5, a] = 1$ since both 12 and 15 are edges of E^-.
- Edge 14 of E^+ leads to $\boldsymbol{x}[4] \leftarrow a$ and no new values in F, since vertex 4 is isolated in E^-.
- Edge 24 of E^+ requires a new letter because $F[2, a] = 1$. Therefore we assign $\boldsymbol{x}[2] \leftarrow b$ and $\boldsymbol{x}[4] \overset{+}{\leftarrow} b$, while setting $F[1, b] = F[5, b] = 1$ because of the edges 21 and 25 in E^-.
- Finally we deal with the isolated vertex 5 in E^+ by setting $\boldsymbol{x}[5] \leftarrow c$ since $15 \in E^-$ and $\boldsymbol{x}[1] = a$, while $25 \in E^-$ and $\boldsymbol{x}[2] = b$.

The lexicographically least string is $\boldsymbol{x} = aba\{a, b\}c$.

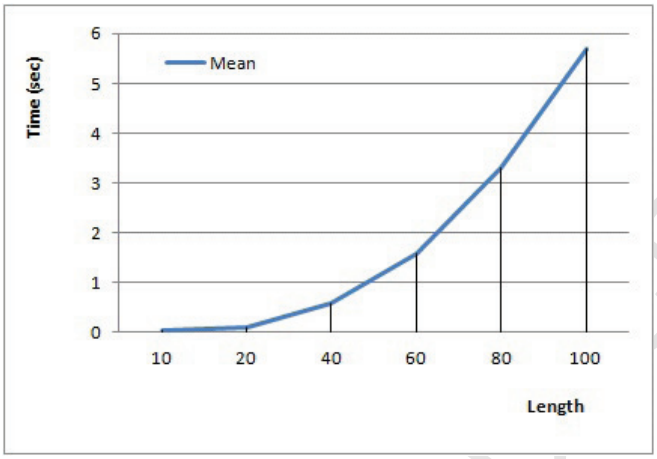

Fig. 7. Timing results for randomly-generated feasible arrays $\boldsymbol{y}$.

3.5 Computational Experiments

To get an idea of how the algorithm behaves in practice, we have implemented Algorithm RevEng and conducted a simple experimental study. A set of 1000 feasible arrays having lengths $10, 20, .., 100$ has been randomly generated as follows. For each feasible array $\boldsymbol{y}$ we randomly select a value for $\boldsymbol{y}[i], i \in [1..n]$ from within the range $[0..n - i + 1]$. The experiments have been run 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. We have implemented Algorithm RevEng in $C\#$ language using Visual Studio 2010. As is evident from Figure 7, the experiments suggest that average case time also increases by a factor somewhat greater than n^2.

4 Discussion

The high worst-case time complexity of the algorithm described here suggests room for improvement. On the other hand, it is difficult to imagine an algorithm that could do the same computation without considering all the edges of E^+ and thus necessitating $\Theta(n^2)$ time for many instances of the prefix table $\boldsymbol{\pi}$. Similarly, the requirement to achieve a lexicographically least solution leads to a recurring dependence on alphabet size σ that expresses itself in the time complexity. Even though it may be true that $\sigma \leq n$, nevertheless it seems clear that σ can be much larger than in the regular case, where it has been shown [7, 6] that $\sigma \leq \lceil \log_2 n \rceil$.

We have tried approaches that focus on E^- rather than E^+ as the primary data structure, but without success. In particular, we have considered "triangles" $i_1 j i_2$, where both (i_1, j) and (i_2, j) are edges in E^+, while $(i_1, i_2) \in E^-$, a situation that forces a string to be indeterminate. It turns out, however, that the number of such triangles is $O(n^2)$. Similarly, the ingenious graph proposed in [4], whose chromatic number is the minimum alphabet size σ of a string corresponding to a given prefix table, has $O(n^2)$ vertices in the worst case.

At the same time, we have no proof that our algorithm is asymptotically optimal; for example, an algorithm that could eliminate the σ factor in the complexity would be of considerable interest. Also interesting would be an algorithm for indeterminate strings that would execute in $\Theta(n)$ time on regular strings as a special case, thus matching the algorithm of [7]. More generally, we propose the study of indeterminate strings ("strings" as we have called them here), their associated data structures (such as the prefix table), and their applications as a promising research area in both combinatorics on words and string algorithms.

References

1. K. Abrahamson. Generalized string matching. *SIAM J. Comput.*, 16(6):1039–1051, 1987.
2. Hideo Bannai, Shunsuke Inenaga, Ayumi Shinohara, and Masayuki Takeda. Inferring strings from graphs and arrays. In Branislav Rovan and Peter Vojt's, editors, *Symp. on Mathematical Foundations of Computer Science*, volume 2747 of *Lecture Notes in Computer Science*, pages 208–217. Springer, 2003.
3. Francine Blanchet-Sadri. *Algorithmic Combinatorics on Partial Words*. Chapman & Hall CRC, 2008.
4. Francine Blanchet-Sadri, Michelle Bodnar, and Benjamin De Winkle. New bounds and extended relations between prefix arrays, border arrays, undirected graphs, and indeterminate strings. In N. Portier and E. Mayr, editors, *Proc. 31st Symp. on Theoretical Aspects of Computer Science*, pages 162–173, 2014.
5. Widmer Bland, Gregory Kucherov, and W. F. Smyth. Prefix table construction and conversion. *Proc. 24th Internat. Workshop on Combinatorial Algs. (IWOCA)*, pages 41–53, 2013.
6. Manolis Christodoulakis, P. J. Ryan, W. F. Smyth, and Shu Wang. Indeterminate strings, prefix arrays and undirected graphs. *CoRR abs/1406.3289*, 2014.
7. Julien Clément, Maxime Crochemore, and Giuseppina Rindone. Reverse engineering prefix tables. In *Proc. 26th Symp. on Theoretical Aspects of Computer Science*, pages 289–300, 2009.
8. Jean-Pierre Duval, Thierry Lecroq, and Arnaud Lefebvre. Border array on bounded alphabet. *J. Automata, Languages & Combinatorics*, 10(1):51–60, 2005.
9. M.J. Fischer and M.S. Paterson. String matching and other products. In R.M. Karp, editor, *Complexity of Computation,*, pages 113–125. American Mathematical Society, 1974.

10. Frantisek Franek, Shudi Gao, Weilin Lu, P. J. Ryan, W. F. Smyth, Yu Sun, and Lu Yang. Verifying a border array in linear time. *J. Comb. Math. Comb. Comput*, 42:223–236, 2000.
11. Frantisek Franek, Weilin Lu, P. J. Ryan, W. F. Smyth, Yu Sun, and Lu Yang. Verifying a border array in linear timee (preliminary version). *Proc. 10th Australasian Workshop on Combinatorial Algs. (AWOCA) School of Computing, Curtin University of Technology*, pages 26–33, 1999.
12. Frantisek Franek and William F. Smyth. Reconstructing a suffix array. *Int. J. Found. Comput. Sci.*, 17(6):1281–1296, 2006.
13. Jan Holub and W. F. Smyth. Algorithms on indeterminate strings. *Proc. 14th Australasian Workshop on Combinatorial Algs. (AWOCA)*, pages 36–45, 2003.
14. Costas S. Iliopoulos, Manal Mohamed, Laurent Mouchard, Katerina Perdikuri, William F. Smyth, and Athanasios K. Tsakalidis. String regularities with don't cares. *Nordic J. Comput.*, 10(1):40–51, 2003.
15. Costas S. Iliopoulos, Laurent Mouchard, and M. Sohel Rahman. A new approach to pattern matching in degenerate DNA/RNA sequences and distributed pattern matching. *Math. in Computer Science*, pages 557–569, 2008.
16. Michael G. Main and Richard J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *J. Algorithms 5*, pages 422–432, 1984.
17. Dennis Moore, W. F. Smyth, and Dianne Miller. Counting distinct strings. *Algorithmica 23(1)*, pages 1–13, 1999.
18. Tanaeem M. Moosa, Sumaiya Nazeen, M. Sohel Rahman, and Rezwama Reaz. Linear time inference of strings from cover arrays using a binary alphabet. *Disc. Math., Algs. & Appls. 5(2)*, pages 160–172, 2013.
19. Sumaiya Nazeen, Sohel M. Rahman, and Rezwana Reaz. Indeterminate string inference algorithms. *J. Discrete Algorithms*, 10:23–34, 2012.
20. Bill Smyth. *Computing Patterns in Strings*. Pearson/Addison–Wesley, 2003.
21. W. F. Smyth and Shu Wang. New perspectives on the prefix array. *Proc. 15th String Processing & Inform. Retrieval Symp. (SPIRE)*, 5280:133–143, 2008.