Symposium on Theoretical Aspects of Computer Science 2009 (Freiburg), pp. 289–300
www.stacs-conf.org

REVERSE ENGINEERING PREFIX TABLES

JULIEN CLÉMENT [1] AND MAXIME CROCHEMORE [2,3] AND GIUSEPPINA RINDONE [3]

[1] GREYC, CNRS-UMR 6072, Université de Caen, 14032 Caen, France

[2] King's College London, Strand, London WC2R 2LS, UK

[3] Institut Gaspard-Monge, CNRS-UMR 8049, Université de Paris-Est
E-mail address: clement@info.unicaen.fr,maxime.crochemore@kcl.ac.uk,rindone@univ-mlv.fr

Abstract. The Prefix table of a string reports for each position the maximal length of its prefixes starting here. The Prefix table and its dual Suffix table are basic tools used in the design of the most efficient string-matching and pattern extraction algorithms. These tables can be computed in linear time independently of the alphabet size.

We give an algorithmic characterisation of a Prefix table (it can be adapted to a Suffix table). Namely, the algorithm tests if an integer table of size n is the Prefix table of some word and, if successful, it constructs the lexicographically smallest string having it as a Prefix table. We show that the alphabet of the string can be bounded to $\log_2 n$ letters. The overall algorithm runs in $O(n)$ time.

1. Introduction

The Prefix table of a string reports for each position on the string the maximal length of its prefixes starting at that position. The table stores the same information as the Border Table of the string, which memorises for each position the maximal length of prefixes ending at that position. Both tables are useful in several algorithms on strings. They are used to design efficient string-matching algorithms and are essential for this type of applications (see for example [15] or [6]).

The dual notion of the Prefix table, the Suffix Table (not to be confused with the Suffix Array that is related to the lexicographic order of the suffixes), simplifies the design of the pattern preprocessing for Boyer-Moore string-matching algorithm ([4], see also [6]). This preprocessing problem is notorious for its several unsuccessful attempts. Gusfield makes it a fundamental element of string-matching methods he presents in [15]. His Z-algorithm corresponds to the computation of the Suffix table. Along the same line, the Suffix table is an essential element of the Apostolico-Giancarlo string-matching algorithm [1], one of the ultimate improvements of the Boyer-Moore strategy: the maximal number of symbol comparisons is no more than $3n/2$ in the worst-case [9], which is half the number

Key words and phrases: design and analysis of algorithms, algorithms on strings, pattern matching, string matching, combinatorics on words, Prefix table, Suffix table.

This work is funded partially by CNRS and by the European Project AutoMathA.

of the original searching algorithm, and this is linked to the total running time of the algorithm. By the way, the technique is still the object of a conjecture about the polynomial number of configurations encountered during a search (see [17, 2]). These techniques or some reductions of them are the best choice for implementing search tools inside text editors or analogue software.

The computation of the table of borders is a classical question. It is inspired by an algorithm of Morris and Pratt (1970), which is at the origin of the first string-matching algorithm running in linear time independently of the alphabet size [17]. Despite the fact that Border Tables and Prefix tables contains the same information on the underlying string, the efficient algorithms to compute them are quite different (see [6, Chapter 1]). The first algorithm has a recursive expression while this is not so for the second one.

The technique used to compute the Prefix table works online on the string and takes advantage on what has been done on shorter prefixes of the string. It needs only simple combinatorial properties to be proved. A similar technique is used by Manacher [18] for finding the shortest even palindrome prefix of a string, which extends to a linear-time decomposition of a string into even palindromes ([14], see also [17]).

In this article we consider the reverse engineering problem: testing if a table of integers is the Prefix table of some string, and if so, producing such a string.

The same question can be stated for other data structures storing information on strings. Successful solutions then provide if-and-only-if conditions on the data structures and then complete characterisations, which is of main theoretical interest. It is also of interest for software testing problem related to the data structures when it comes to implementing them. It helps also design methods for randomly generating the data structures in relation to the previous problem.

As far as we know, reverse engineering has been first considered for Border Tables by Franek et al. [12]. Their algorithm tests whether an integer array is the Border Table of a string or not, and exhibits a corresponding string if it is. They solve the question in linear time, according to the size of the input array, for an unbounded alphabet. A refinement of the technique by Duval et al. [10, 11] solves the question for a bounded-size alphabet. Bannai et al. [3] characterise three other data structures intensively used in String Algorithmics: Directed Acyclic Subsequence Graph, Directed Acyclic Word Graph or Minimal Suffix Automaton, and Suffix Array. Their three testing algorithms run in linear time as well. Reconstructing a Suffix Array is obvious on a binary alphabet especially if a special symbol is appended to the string. For general alphabets, another solution is by Franek et al. [13]. In this situation, the algorithm accounts for descents in permutations (see [5]).

The case of Prefix table (or Suffix table) seems more difficult. However we get, as for previous questions, a linear-time test. The algorithm can provide the largest initial part of the array that is compatible with a Prefix table. When the array is a Prefix table it infers a string corresponding to it on the smallest possible alphabet. We show that indeed $\log_2 n$ letters are enough for a table of size n. In the algorithm we make use of a variable to store a set of letters that are forbidden in a certain context. It is surprising and remarkable that, due to combinatorial properties, the algorithm requires no sophisticated data structure to implement this variable and still runs in linear time.

Reverse engineering for the Longest Previous Factor table, which is an important component of Ziv-Lempel text compression method (see [7, 8]), is an open question. It is not known if a linear-time solution exists.

In next sections we first describe properties of Prefix tables, then design the testing algorithm, and finally analyse it.

2. Properties of Prefix tables

After basic definitions we state the major properties of Prefix tables.

2.1. Preliminaries and definitions

Let A be an ordered alphabet, $A = \{a_0, a_1, \ldots\}$. A word w of length $|w| = n$ is a finite sequence $w[0]w[1]\ldots w[n-1] = w[0\,..\,n-1]$ of letters of A. The language of all words is A^*, and A^+ is the set of nonempty words. For $0 \le i, j < n$ we say that the factor $w[i\,..\,j]$ occurs at position i on w. By convention, $w[i\,..\,j]$ is the empty word ε if $i > j$. The prefix (resp. suffix) of length ℓ, $0 \le \ell \le n$, of w is the word $u = w[0\,..\,\ell-1]$ (resp. $u = w[n-\ell\,..\,n-1]$). A border u of w is a word that is both a prefix and a suffix of w distinct from w itself.

Definition 2.1 (Prefix table)**.** The Prefix table Pref_w of a word $w \in A^+$ of length n, is the array of size n defined, for $0 \le i < n$, by

$$\mathrm{Pref}_w[i] = \mathrm{lcp}(w, w[i\,..\,n-1]),$$

where $\mathrm{lcp}(u, v)$ denotes the length of the longest common prefix of two words u and v.

Definition 2.2 (Extent and Anchor)**.** On a table t of size n we define, for $0 < i \le n$:

- the (right) *extent* $\mathrm{Extent}(t, i)$ of position i (according to t) is the integer
$$\mathrm{Extent}(t, i) = \max\{j + t[j] \mid 0 < j < i\} \cup \{i\},$$
- the *anchor* $\mathrm{Anchor}(t, i)$ of position i (according to t) is the set of positions on t reaching the extent of i, i.e.,
$$\mathrm{Anchor}(t, i) = \{f \mid 0 < f < i \text{ and } f + t[f] = \mathrm{Extent}(t, i)\}.$$

Remark 2.3. Let us consider two positions i and j on the Prefix table t, $i < j$, that satisfy $\mathrm{Extent}(t, i) \neq \mathrm{Extent}(t, j)$. Then, $\mathrm{Anchor}(t, i) \cap \mathrm{Anchor}(t, j) = \emptyset$. Indeed in this case $\mathrm{Extent}(t, i) < \mathrm{Extent}(t, j)$, and we have $\max \mathrm{Anchor}(t, i) < i$ while $\min \mathrm{Anchor}(t, j) \ge i$.

In the description of the algorithm and its analysis we refer to the set of letters following immediately the prefix occurrences of the borders of a word w. For $0 < i \le k \le n$, we note

$$\mathcal{E}(w, i, k) = \{w[|u|] \mid u \text{ is a border of } w[0\,..\,k-1] \text{ and } |u| > k - i\},$$

and we simply write $\mathcal{E}(w, k)$ when considering $\mathcal{E}(w, k, k)$.

Example. Let w be the word `ababaababababa`. The following table shows its Prefix table t, the values $\mathrm{Extent}(t, i)$, and the sets $\mathrm{Anchor}(t, i)$, for all values of i.

i	0	1	2	3	4	5	6	7	8	9	10	11	12
$w[i]$	a	b	a	b	a	a	b	a	b	a	b	a	–
$t[i]$	12	0	3	0	1	5	0	5	0	3	0	1	–
$\mathrm{Extent}(t, i)$	–	1	2	5	5	5	10	10	12	12	12	12	12
$\mathrm{Anchor}(t, i)$	–	$\emptyset$	$\emptyset$	$\{2\}$	$\{2\}$	$\{2,4\}$	$\{5\}$	$\{5\}$	$\{7\}$	$\{7\}$	$\{7,9\}$	$\{7,9\}$	$\{7,9,11\}$

The whole word has three nonempty borders: `ababa`, `aba`, and `a` occurring at respective positions 7, 9, and 11. We have $\mathcal{E}(w, 9, 12) = \{\mathtt{a}\}$ because $u = \mathtt{ababa}$ is the only border of w of length larger than $3 = 12 - 9$ and $w[|u|] = \mathtt{a}$. We have $\mathcal{E}(w, 12) = \{\mathtt{a}, \mathtt{b}\}$ because

the three nonempty borders of w are `ababa`, `aba`, and `a`, and because $w[5] = \{\texttt{a}\}$ and $w[3] = w[1] = \{\texttt{b}\}$.

Definition 2.4 (Validity and compatibility)**.** Let t be a table of size n. For $0 < i < n$, we say that:

- the table t is *valid until* i if there exists a word w for which $t[1\,..\,i] = \mathrm{Pref}_w[1\,..\,i]$. We also say that t is *compatible with* w *until* i.
- The table t is *valid* if there exists a word w such that $t = \mathrm{Pref}_w$ (note that in particular $t[0] = n$). We also say that t is *compatible with* w.

Example. Let t and t' be the two tables defined by:

i	0	1	2	3	4	5
$t[i]$	6	0	0	2	0	1
$t'[i]$	6	0	0	2	1	1

The table t is valid since it is compatible with `abcaba` for example. But the table t' is not valid since it cannot be the Prefix table of any word. However, t' is compatible with `abcaba` until position 3.

2.2. Properties

In this subsection we state if-and-only-if conditions on values appearing in a Prefix table. The algorithm of the next section is derived from them.

By definition, the values in the Prefix table of a word w of length n satisfy

$$0 \le \mathrm{Pref}_w[i] \le n - i \text{ and } \mathrm{Pref}_w[0] = n\ .$$

There is another simple property coming from the definitions of Extent and $\mathcal{E}$: when $g = \mathrm{Extent}(\mathrm{Pref}_w, i) = i$ and $i < n$, $w[i] \notin \mathcal{E}(w, g)$.

The next statement gives less obvious conditions (some of them are in [6]).

Proposition 2.5 (Necessary and sufficient conditions on values in a prefix table)**.** *Let w be a word of length n and Pref_w its Prefix table. Let $i \in \{1, \ldots, n-1\}$ and $g = \mathrm{Extent}(\mathrm{Pref}_w, i)$, and assume $i < g$. Then, for all $f \in \mathrm{Anchor}(\mathrm{Pref}_w, i)$, one has*

(i) $\mathrm{Pref}_w[i] < g - i$ *if and only if* $\mathrm{Pref}_w[i - f] < g - i$ *and then both* $\mathrm{Pref}_w[i] = \mathrm{Pref}_w[i - f]$ *and* $w[i - f + \mathrm{Pref}_w[i]] = w[i + \mathrm{Pref}_w[i]] \neq w[\mathrm{Pref}_w[i]]$.

(ii) $\mathrm{Pref}_w[i] = g - i$ *if and only if* $\begin{cases} \mathrm{Pref}_w[i - f] \ge g - i \text{ and} \\ g = n \text{ or } w[g] \neq w[g - i]\ . \end{cases}$

(iii) $\mathrm{Pref}_w[i] = g - i + \ell$ *and* $\ell > 0$ *if and only if*

$$\begin{cases} \mathrm{Pref}_w[i - f] = g - i \text{ and} \\ w[g\,..\,g + \ell - 1] = w[g - i\,..\,g - i + \ell - 1] \text{ and} \\ g + \ell = n \text{ or } w[g + \ell] \neq w[g - i + \ell]\ . \end{cases}$$

Proof. Let $f \in \mathrm{Anchor}(\mathrm{Pref}_w, i)$. Let $u = w[f\,..\,g-1]$ and $v = w[i\,..\,g-1]$. By definition of f and g, the word u is the longest prefix of w beginning at position f. This ensures that $w[g] \neq w[g - f]$ if $g < n$. The suffix v of u has another occurrence at position $i - f$ and one has $|v| = g - i$. With the notation, we get the following.

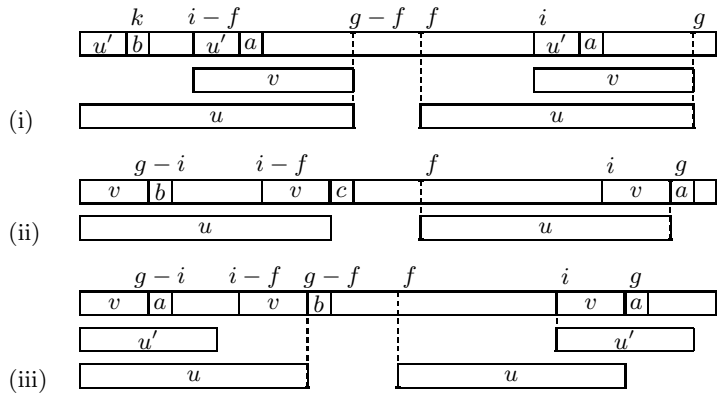

Figure 1: Illustration for the proof of Proposition 2.5. (**i**) Case $\mathrm{Pref}_w[i] < g - i$. In word w, the letter following u' at position $i - f$ is the same as the letter following it at i, and different from the letter at position $k = \mathrm{Pref}_w[i]$ ($a \neq b$). (**ii**) Case $\mathrm{Pref}_w[i] = g - i$. In word w, we have $\mathrm{Pref}_w[i - f] \geq g - i$. If $w[g]$ is defined and equals a, we have both $a \neq b$ and $a \neq c$. If in addition $b \neq c$, then $\mathrm{Pref}_w[i - f] = g - i$, otherwise, $\mathrm{Pref}_w[i - f] > g - i$. (**iii**) Case $\mathrm{Pref}_w[i] > g - i$. In word w, we have $a \neq b$ and then $\mathrm{Pref}_w[i - f] = g - i$.

(i) When $\mathrm{Pref}_w[i] = k < g - i = |v|$, one has $\mathrm{Pref}_w[i - f] = k$ (and the converse is also true). Moreover, $w[i\,..\,i + k - 1] = w[i - f\,..\,i - f + k - 1] = w[0\,..\,k - 1]$ and $w[i + k] = w[i - f + k] \neq w[k]$ (see Figure 1(i)).

(ii) Suppose $\mathrm{Pref}_w[i] = g - i$. Then $w[g] \neq w[g - i]$, if $g \neq n$, by definition of Pref_w. The word v being a suffix of u also occurs at position $i - f$, which implies $\mathrm{Pref}_w[i - f] \geq g - i$ as expected. Conversely, if $\mathrm{Pref}_w[i - f] \geq g - i$ then v is a prefix of w that occurs at position i. Since $w[g] \neq w[g - i]$ when $g < n$, one has $\mathrm{Pref}_w[i] = |v| = g - i$ (see Figure 1(ii)). The conclusion holds obviously if $g = n$.

(iii) Suppose $\mathrm{Pref}_w[i] = g - i + \ell$ with $\ell > 0$. Let $v' = w[g\,..\,g + \ell - 1]$. We have by definition of Pref_w, $v = w[0\,..\,g - i - 1]$, $v' = w[g - i\,..\,g - i + \ell - 1]$, and $w[g + \ell] \neq w[g - i + \ell]$ (if $g + \ell < n$). So one has $w[g] = w[g - i]$ (first letter of v') and since $w[g] \neq w[g - f]$, we get $\mathrm{Pref}_w[i - f] = |v| = g - i$. The converse is also true by definition of Pref_w since $v = w[i\,..\,g - 1] = w[i - f\,..\,g - f - 1] = w[0\,..\,g - i - 1]$ (see Figure 1(iii)). ∎

The next proposition ensures that $\mathrm{Pref}_w[1\,..\,i]$ is only determined by the prefix of the word of length $\mathrm{Extent}(\mathrm{Pref}_w, i + 1)$.

Proposition 2.6. *Let $w = w[0\,..\,n - 1]$ be a word of length n and Pref_w its Prefix table. Let $i \in \{1, \ldots, n - 1\}$ and $g = \mathrm{Extent}(\mathrm{Pref}_w, i)$. Let x be the prefix of w of length g. Then $\mathrm{Pref}_w[1\,..\,i - 1] = \mathrm{Pref}_x[1\,..\,i - 1]$.*

Proof. By definition of g, we know that prefixes of w starting at j, $1 \leq j < i$, do not end beyond position $g - 1$. The letter at position g acts as a marker for all these prefixes, whence the equality. ∎

3. Testing an integer table

In this section, we describe the testing algorithm and prove its correctness.

3.1. Algorithm

The algorithm TableToWord takes as input an integer array t of size n, and checks whether t is a valid Prefix table. If so, the output is the smallest (according to the lexicographic order) word w for which $t = \mathrm{Pref}_w$ together with the number of distinct symbols needed. If t is not valid, the algorithm exits with error. It can be modified to output the longest initial part of the table that is valid.

```
TableToWord(t, n)
 1  if t[0] ≠ n then
 2        return Error("incompatible length")
 3  w[0] ← first symbol of the alphabet
 4  k ← 1
 5  g ← 1
 6  E ← ∅
 7  for i ← 1 to n − 1 do
 8        if t[i] < 0  or  t[i] > n − i then
 9              return Error("error at position", i)
10        if i = g then
11              if t[i] = 0 then
12                    w[g] ← ChooseNotIn(E ∪ {w[0]})
13                    if w[g] is a new letter then
14                          k ← k + 1
15                    E ← ∅
16                    g ← g + 1
17              elseif w[0] ∈ E then
18                    return Error("error at position", i)
19              else Copy(w, i, 0, t[i])
20                    (f, g) ← (i, i + t[i])
21                    E ← {w[g − f]}
22        elseif t[i] < g − i then                          ▷ Case (i)
23              if t[i − f] ≠ t[i] then
24                    return Error("error at position", i)
25        elseif t[i] = g − i then                          ▷ Case (ii)
26              if t[i − f] < g − i then
27                    return Error("error at position", i)
28              E ← E ∪ {w[g − i]}
29        elseif t[i − f] ≠ g − i  or  w[g − i] ∈ E then    ▷ Case (iii)
30              return Error("error at position", i)
31        else Copy(w, g, g − i, t[i] − g + i)
32              (f, g) ← (i, i + t[i])
33              E ← {w[g − f]}
34  return (w, k)
```

The function Copy is defined as follows: $\mathrm{Copy}(w, i, j, \ell)$ copies successively ℓ letters at positions $j, j+1, \ldots, j+\ell-1$ to respective positions $i, i+1, \ldots, i+\ell-1$. Note that some letters may be undefined when the process starts.

The algorithm is essentially built from the properties of Prefix tables stated in Proposition 2.5 and before the statement. The variable E implements the set $\mathcal{E}$ and stores the letters that should not appear at position g in a valid table. In the text of the algorithm,

the function ChooseNotIn(S) is used to return the first symbol which is not in the set S. Implementations of E and ChooseNotIn are discussed in Section 4.

The correctness of the algorithm is established in Propositions 3.2 and 3.3 below.

3.2. Correctness of the algorithm

Before proving that the algorithm TableToWord is correct in Propositions 3.2 and 3.3, we establish a lemma consisting in three parts. This lemma is closely related to Proposition 2.5 on prefix tables. However, the next lemma is concerned with an integer table t only known to be valid until $i-1$ and gives conditions on the next value $t[i]$.

Lemma 3.1. *In the following three statements, t is an integer table of size n, index $i \in \{1,\dots,n-1\}$ is fixed, $g = \mathrm{Extent}(t,i)$, and $\mathcal{A} = \mathrm{Anchor}(t,i)$. The table is assumed to be valid until $i-1$, compatible with the word w until $i-1$. We also suppose $i < g$. The following properties hold:*

a) *If $t[i] < g-i$ and if there exists $f \in \mathcal{A}$ such that $t[i-f] = t[i]$, then for all $f' \in \mathcal{A}$, $t[i-f'] = t[i]$.*
b) *If $t[i] = g-i$ and there exists $f \in \mathcal{A}$ such that $t[i-f] \geq g-i$, then for all $f' \in \mathcal{A}$, $t[i-f'] \geq g-i$.*
c) *If $t[i] > g-i$ and there exists $f \in \mathcal{A}$ such that $t[i-f] = g-i$ and $w[g-i] \notin \mathcal{E}(w,i,g)$, then for all $f' \in \mathcal{A}$, $t[i-f'] = g-i$.*

Proof. The proofs for the three properties rely on the same kind of arguments, but for lack of space we only detail the first one.

Let $f \in \mathcal{A}$ be such that $t[i-f] = t[i]$ and let any $f' \in \mathcal{A}$. Since t is compatible with w until $i-1$, we have both $t[f] = \mathrm{Pref}_w[f] = g-f$ and $t[f'] = \mathrm{Pref}_w[f'] = g-f'$. Let $y = w[i\,..\,i+t[i]-1]$ and $v = w[i\,..\,g-1]$. By hypothesis, y is a proper prefix of v, and the word v occurs at positions $i-f$ and $i-f'$. So, letters at positions $i+t[i]$, $i-f+t[i]$ and $i-f'+t[i]$ are the same. And this letter is different from letter at position $t[i]$ since $\mathrm{Pref}_w[i-f] = t[i]$. Therefore $t[i-f'] = \mathrm{Pref}_w[i-f'] = t[i]$. ∎

Proposition 3.2 (Correctness of the algorithm)**.** *Suppose the algorithm terminates normally (i.e., without error). At the beginning of the ith iteration,, we have the following relations for the set of variables $\{w, f, g, E\}$ of the algorithm :*

(1) $g = \mathrm{Extent}(t,i)$;
(2) *The word w built is of length $|w| = g$;*
(3) $f = \min \mathrm{Anchor}(t,i)$ *if* $i < g$;
(4) $E = \mathcal{E}(w,i,g)$;
(5) $t[1\,..\,i-1] = \mathrm{Pref}_w[1\,..\,i-1]$.

Proof. The proof is based upon recursion on the index i.

Properties 1–4 are easy to check. When $i = g$, the last property is a direct consequence both of Pref and Extent definitions and of Proposition 2.6. When $i < g$, it is a direct consequence of Lemma 3.1 as well as of propositions 2.5 and 2.6. ∎

Proposition 3.3. *If the algorithm exits with error at the ith iteration on input table t, then t is valid until position $i-1$ but not until i.*

Proof. The table is trivially invalid if the algorithm stops on line 2 or line 9. So we now suppose that $t[0] = n$ and also that integers in the table are never out of bounds.

Suppose we have an error during the ith iteration with $i > 0$. Let w be the word built so far at the beginning of the ith iteration. Thanks to Proposition 3.2, t is compatible with w until position $i-1$, so that t is also valid until position $i-1$, and we have $g = \text{Extent}(t, i)$, $f \in \text{Anchor}(t, i) = \mathcal{A}$, $E = \mathcal{E}(w, i, g)$.

We examine several cases depending on which line of the algorithm the error is produced.

Line 18. One has $i = g$, $t[i] > 0$, $E = \mathcal{E}(w, i, g) = \mathcal{E}(w, g)$ and $w[0] \in E$. There exists $f' \in \mathcal{A}$ such that $w[g - f'] = w[0]$. Thus, $t[g - f'] = \text{Pref}_w[g - f'] > 0$. By contradiction, suppose now that there exists a word z such that $t[1\,..\,i] = \text{Pref}_z[1\,..\,i]$. We must have $\text{Pref}_z[i] = t[i] > 0$ and then for all $f \in \mathcal{A}$, $z[g - f] \neq z[0]$. Thus $\text{Pref}_z[g - f] = 0$, for all $f \in \mathcal{A}$ and it is true in particular for f'. So we have $\text{Pref}_z[g - f'] = 0 \neq t[g - f']$, against the hypothesis.

Line 24 or 27. The entry $t[i]$ does not satisfies one of the properties stated in Proposition 2.5.

Line 30. We have $t[i] > g - i$. If $t[i - f] \neq g - i$, then $t[i]$ is in contradiction with Proposition 2.5 and is invalid. Suppose now that $t[i - f] = g - i$ and $w[g - i] \in E$. As in the previous case (on Line 18), there exists $f' \in \mathcal{A}$ such that $w[g - f'] = w[g - i]$ and $\text{Pref}_w[i - f'] = t[i - f'] > g - i$. Again, by contradiction, suppose that there exists a word z such that $t[1\,..\,i] = \text{Pref}_z[1\,..\,i]$. We must have $\text{Pref}_z[i] = t[i] > g - i$. By Proposition 2.5, we get $\text{Pref}_z[i - f] = g - i$, for all $f \in \mathcal{A}$, and in particular that $\text{Pref}_z[i - f'] = g - i$. Thus $\text{Pref}_z[i - f'] \neq t[i - f']$, which yields a contradiction.

As a conclusion, if the algorithm stops with error during the ith iteration, there is no word z such that $t[1\,..\,i] = \text{Pref}_z[1\,..\,i]$, which means that the table is valid until $i-1$ only. ∎

As an immediate consequence of the last two propositions we get the following statement.

Corollary 3.4. *An integer table t is a valid Prefix table if and only if the algorithm* TableToWord *terminates without error. Moreover the output word w it produces is the lexicographically smallest word compatible with t.*

4. Analysis of the algorithm

After the correctness of the algorithm TableToWord, the present section is devoted to its complexity analysis in the worst case. We first consider its running time under some assumption (proved later), show a remarkable property of contiguity of forbidden letters, and then evaluate the number of letters required to built the smallest word associated with a valid table.

4.1. Linear running time

We assume that the variable E of algorithm TableToWord is implemented with a mere linear list (linked or not). Although the variable is associated with a set of letters, we allow several copies of the same letter in the list. Therefore, adding a letter to the list or initialising it to the empty set can be performed in constant time. Membership is done in time $O(|E|)$, denoting by $|E|$ the length of the list.

In this subsection, we additionally assume that ChooseNotIn, the choice function, executes in constant time. There is no problem to get this condition if we do not require the algorithm produces the smallest word associated with a valid table. But even with

this requirement we show how to satisfy the hypothesis in Section 4.2, using a contiguity property of letters in E.

Proposition 4.1 (Time complexity)**.** *The algorithm* TableToWord *can be implemented to run in time $O(n)$ on an input integer table of size n.*

Proof. Let t be the input table. The only non constant-time instructions inside the for loop of the algorithm are copies of words (lines 19 and 31) and operations on the set of letters via the variable E (lines 6, 12, 15, 17, 21, 28, 29, and 33).

Let w be the word built by the algorithm. We note that letters of w are assigned to positions exactly once. Hence the total number of copies of letter is exactly n.

Initialisation and update operations can both be achieved in constant time with the considered implementation of E. As for membership tests at lines 17 and 29, the number of letters we have to test is at most the number of elements in $\mathrm{Anchor}(t, i)$. But we note that the value of $g = \mathrm{Extent}(t, i)$ is strictly increased eventually after the test, which means that $\mathrm{Extent}(t, i+1) > \mathrm{Extent}(t, i)$. Since by Remark 2.3 in this situation $\mathrm{Anchor}(t, i)$ is disjoint from all the $\mathrm{Anchor}(t, j)$, $j > i$, the number of letter comparisons is less than n (number of positions, excluding position 0).

It remains to consider the time to choose a letter at line 12 with function ChooseNotIn, which we assume temporarily that it is constant. We can thus conclude to an overall linear running time. ∎

4.2. Contiguous letters

We show in this section that the set of letters stored in the variable E of the algorithm TableToWord satisfies a contiguity condition. This property leads to a very simple and efficient implementation of the function ChooseNotIn, which meets the requirement stated before Proposition 4.1. We start with the extra definition of a choice position: it is a position on the table t for which the algorithm has to choose a letter.

Definition 4.2 (Choice position)**.** Let t be an integer table of size n. A position g on t is called a choice position if $\mathrm{Extent}(t, g) = g$ (that is, if for all $j < g$, $j + t[j] \leq g$) and $t[g] = 0$.

Remark. If g is a choice position, all borders of $w[0\,..\,g-1]$ (t compatible with w) are *strict borders* (if the border has length k, $w[k] \neq w[g]$). It is known that the number of strict borders of a word of length n is at most $\log_\varphi n$ where φ is the golden mean (see for example [6, Page 91]). This suggests that the alphabet size of a word admitting t as Prefix table is logarithmic in the length of t. The next section shows the logarithmic behaviour more accurately with a different argument.

Proposition 4.4 below states that each time the function ChooseNotIn is called on line 12 of the algorithm TableToWord the set $\mathcal{E}(w, g) \cup \{w[0]\}$ is a set of contiguous letters, that is, it contains all the letters between a_0 and the largest one in $\mathcal{E}(w, g) \cup \{w[0]\}$.

We introduce convenient definitions and notation related to a prefix u of the word w:

- $\mathit{Next}(u) = \{w[|v|] \mid v \in A^* \text{ border of } u\}$ (note that $\mathit{Next}(\varepsilon) = \emptyset$).
- For $a \in \mathit{Next}(u)$, the *special border* of u with respect to letter a is the shortest border v such that $w[|v|] = a$.

Lemma 4.3. *Let g be a choice position on Pref_w, and let v, $v \neq \varepsilon$, be the special border of $w[0\,..\,g-1]$ with respect to some letter b. Then $|v|$ is also a choice position.*

Proof. Since $v \neq \varepsilon$ and v is a special border, we have $b \neq w[0]$, and then $\mathrm{Pref}_w[|v|] = 0$. By contradiction, suppose $|v|$ is not a choice position. There exists a position j, $0 < j < |v|$, such that $j + \mathrm{Pref}_w[j] > |v|$. Let $y = w[j \,..\, |v|-1]$, which is a border of v, and also a border of $w[0 \,..\, g-1]$. But we also have $b = w[|y|]$ and $|y| < |v|$. So v cannot be the special border with respect to b. ∎

Proposition 4.4 (Contiguous)**.** *Let t be a valid Prefix table and w the word produced by the algorithm* TableToWord *from t. For any choice position g on t, setting $k = \mathrm{card}\, Next(w[0 \,..\, g-1])$, we have $Next(w[0 \ldots g-1]) = \{a_0, \ldots, a_{k-1}\}$.*

Proof. Note that choice positions are exactly positions where we must call the function ChooseNotIn (line 12). In this case we always choose the smallest possible letter.

The proof is based upon a recursion on k. In the following g is always a choice position and we set $S_g = Next(w[0 \,..\, g-1])$ and $w_g = w[0 \,..\, g-1]$ to shorten notation.

If $k = 1$, we have $S_g = \{w[0]\}$ since ε is always a border of w_g.
Suppose that for any choice position h with $\mathrm{card}(S_h) = k-1$, we have $S_h = \{a_0, a_1, \ldots, a_{k-2}\}$. Let g be a choice position such that $\mathrm{card}(S_g) = k$ with $k > 1$. Consider the longest special border u of w_g and let $b = w[|u|]$ be its associated letter. We have $u \neq \varepsilon$ since $\mathrm{card}(S_g) > 1$. The previous lemma entails that $|u|$ is a choice position. But b cannot be in $S_{|u|}$ (otherwise u would not be the special border with respect to b). Since $S_g = S_{|u|} \cup \{b\}$, we must have $\mathrm{card}(S_{|u|}) = k-1$. So by recurrence $S_{|u|} = \{a_0, a_1, \ldots, a_{k-2}\}$. Therefore the letter chosen at position $|u|$ is $b = a_{k-1}$.

This ends the recurrence and the proof. ∎

Implementation of ChooseNotIn. Following Proposition 4.4, a simple way to implement the function ChooseNotIn is to store the largest forbidden letter at position g. Choosing a letter remains to take the next letter because of the contiguity property, which can obviously be done in constant time.

4.3. Alphabet size

We evaluate the smallest number of letters needed to build a word associated with a valid Prefix table. We first adapt some properties on borders of a word derived from results by Hancart [16] (see also [6] or [19] for example). In the following, w is a word of length n over the alphabet A.

For any prefix of u of w, we define

$$deg(u) = \mathrm{card}(Next(u)),$$

and note that it is $\mathrm{card}(\mathcal{E}(w, |u|) \cup \{w[0]\})$ when $u \neq \varepsilon$. We have

$$deg(u) = \begin{cases} 0 & \text{if } u = \varepsilon \\ \mathrm{card}\{v \mid v \text{ is a special border of } u\} & \text{otherwise.} \end{cases}$$

Lemma 4.5. *Let u be a nonempty prefix of w. Any special border v of u satisfies the inequality $|v| < |u|/2$.*

Proof. The word v is a special border with respect to $a = w[|v|]$. We prove the statement by contradiction. Suppose $|v| \geq |u|/2$, and let $k = 2|v| - |u|$ and $y = w[0 \,..\, k-1]$. Then y is a border of v (and also of u, see Figure 2) and $|y| < |v|$. But the equality $w[|y|]$ and $w[|v|]$ contradicts the fact that v is a special border. Thus, the inequality holds. ∎

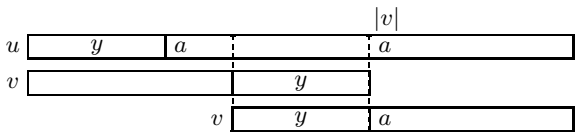

Figure 2: The word y is a border of v and u. Letters at positions $|y|$ and $|v|$ are the same, which prevents v to be a special border of u.

Denoting by $s\text{-}Bord(u)$ the longest special border of u, we get the following corollary.

Corollary 4.6. *Let $w \in A^+$. The length of $s\text{-}Bord(w)$ is smaller than $|w|/2$.*

Lemma 4.7. *Let $w \in A^+$ and u a nonempty prefix of w. One has*

$$deg(u) = deg(s\text{-}Bord(u)) + 1.$$

Proof. This is a direct consequence of the fact that a special border of u is either $s\text{-}Bord(u)$ or a special border of $s\text{-}Bord(u)$. ∎

Proposition 4.8. *For any nonempty prefix u of w, we have*

$$deg(u) \leq \lfloor \log_2(|u| + 1) \rfloor.$$

Proof. The proof relies on a recursion on $|u|$. If $|u| = 1$, then $deg(u) = 1$ since ε is the unique border of u. Thus the property is true. Suppose $|u| > 1$ and that the property holds for any prefix of w of length less than $|u|$. Let us set $v = s\text{-}Bord(u)$. We have

$$deg(u) = deg(v) + 1 \leq \lfloor \log_2(|v| + 1) \rfloor + 1 = \lfloor \log_2(2|v| + 2) \rfloor \leq \lfloor \log_2(|u| + 1) \rfloor.$$

The first equality comes from Lemma 4.7, and the last inequality is obtained *via* Corollary 4.6. ∎

The last lemma yields the following corollary.

Corollary 4.9. *For any nonempty prefix u of w, one has*

$$deg(u) \leq \min\{\mathrm{card}(Alpha(u)), \lfloor \log_2(|u| + 1) \rfloor\},$$

where $Alpha(u)$ is the set of letters occurring in u.

The following example shows that the bound stated in Corollary 4.9 is tight.

Example. We define a sequence of word $(w_i)_{i \geq 0}$ by

$$w_0 = a_0, \text{ and for } i \geq 1 \; w_i = w_{i-1} a_i w_{i-1}.$$

It is straightforward to check that, for $i \geq 0$, we have both $|w_i| = 2^{i+1} - 1$ and $deg(w_i) = \mathrm{card}(Alpha(w_i)) = \lfloor \log_2(|w_i| + 1) \rfloor = i + 1$.

Proposition 4.10. *For the word w produced by the algorithm* TableToWord *from a table of size n, the following (tight) inequality holds*

$$\mathrm{card}(Alpha(w)) \leq \lfloor \log_2(n + 1) \rfloor.$$

Proof. With the same notation as in Proposition 3.2, at the beginning of the ith iteration, the word w of length $g = \mathrm{Extent}(t, i)$ has been built, and we have $E = \mathcal{E}(w, i, g)$. So if $i = g$ and $t[i] = 0$, we have $E \cup \{w[0]\} = Next(w)$. Hence, by Corollary 4.9, we get

$$\mathrm{card}(E \cup \{w[0]\}) = deg(w) \leq \min\{\mathrm{card}(Alpha(w)), \lfloor \log_2(|w| + 1) \rfloor\}.$$

This implies that a new letter must be introduced only if $\mathrm{card}(E \cup \{w[0]\}) = \mathrm{card}(Alpha(w))$. Considering $i = n$ yields the result. ∎

Remark (Size of the maximal alphabet). It is worth noting that a slight modification of the algorithm provides a word with the maximal number of letters and compatible with the input valid table. Indeed we just have, at each call to the function ChooseNotIn, to return a new letter each time it is called. This way we build the word compatible with t with the highest number of letters.

Acknowledgements. We are grateful to Pat Ryan, Bill Smyth, and Shu Wang for interesting discussions on Prefix tables and their extension to wider problems. We thank reviewers for their careful reading of the article and appropriate remarks.

References

[1] A. Apostolico and R. Giancarlo. The Boyer-Moore-Galil string searching strategies revisited. *SIAM J. Comput.*, 15(1):98–105, 1986.

[2] R. Baeza-Yates, C. Choffrut, and G. Gonnet. On Boyer-Moore automata. *Algorithmica*, 12(4/5):268–292, 1994.

[3] H. Bannai, S. Inenaga, A. Shinohara, and M. Takeda. Inferring strings from graphs and arrays. In B. Rovan and P. Vojtás, editors, *Mathematical Foundations of Computer Science*, volume 2747 of *Lecture Notes in Computer Science*, pages 208–217. Springer, 2003.

[4] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[5] M. Crochemore, J. Désarménien, and D. Perrin. A note on the Burrows-Wheeler transformation. *Theoretical Computer Science*, 332(1-3):567–572, 2005.

[6] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, Cambridge, UK, 2007. 392 pages.

[7] M. Crochemore and L. Ilie. Computing Longest Previous Factor in linear time and applications. *Inf. Process. Lett.*, 106(2):75–80, 2008.

[8] M. Crochemore, L. Ilie, and W. F. Smyth. A simple algorithm for computing the Lempel-Ziv factorization. In J. A. Storer and M. W. Marcellin, editors, *18th Data Compression Conference*, pages 482–488. IEEE Computer Society, Los Alamitos, CA, 2008.

[9] M. Crochemore and T. Lecroq. Tight bounds on the complexity of the Apostolico-Giancarlo algorithm. *Information Processing Letters*, 63(4):195–203, 1997.

[10] J.-P. Duval, T. Lecroq, and A. Lefebvre. Border array on bounded alphabet. *Journal of Automata, Languages and Combinatorics*, 10(1):51–60, 2005.

[11] J.-P. Duval, T. Lecroq, and A. Lefebvre. Efficient validation and construction of border arrays. In *Proceedings of 11th Mons Days of Theoretical Computer Science*, pages 179–189, Rennes, France, 2006.

[12] F. Franek, S. Gao, W. Lu, P. J. Ryan, W. F. Smyth, Y. Sun, and L. Yang. Verifying a Border array in linear time. *J. Combinatorial Math. and Combinatorial Computing*, 42:223–236, 2002.

[13] F. Franek and W. F. Smyth. Reconstructing a Suffix Array. *J. Foundations of Computer Sci.*, 17(6):1281–1295, 2006.

[14] Z. Galil and J. I. Seiferas. A linear-time on-line recognition algorithm for "palstar". *J. ACM*, 25(1):102–111, 1978.

[15] D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, Cambridge, UK, 1997. 534 pages.

[16] C. Hancart. On Simon's string searching algorithm. *Inf. Process. Lett.*, 47(2):95–99, 1993.

[17] D. E. Knuth, J. H. M. Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1):323–350, 1977.

[18] G. Manacher. A new linear-time on-line algorithm finding the smallest initial palindrome of a string. *J. Assoc. Comput. Mach.*, 22(3):346–351, 1975.

[19] W. F. Smyth. *Computing Patterns in Strings*. Pearson, Essex, UK, 2003.