Verifying a Border Array in Linear Time

František Franěk Weilin Lu P. J. Ryan W. F. Smyth*
Yu Sun Lu Yang

Algorithms Research Group
Department of Computing & Software
McMaster University
Hamilton, Ontario
Canada L8S 4L7

August 19, 1999

Abstract

A *border* of a string $\boldsymbol{x}$ is a proper (but possibly empty) prefix of $\boldsymbol{x}$ that is also a suffix of $\boldsymbol{x}$. The *border array* $\boldsymbol{\beta} = \boldsymbol{\beta}[1..n]$ of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is an array of nonnegative integers in which each element $\boldsymbol{\beta}[i]$, $1 \le i \le n$, is the length of the longest border of $\boldsymbol{x}[1..i]$. In this paper we first present a simple linear-time algorithm to determine whether or not a given array $\boldsymbol{y} = \boldsymbol{y}[1..n]$ of integers is a border array of some string on an alphabet of unbounded size. We state as an open problem the design of a corresponding and equally efficient algorithm on an alphabet of bounded size α. We then consider the problem of generating all possible distinct border arrays of given length n on a bounded or unbounded alphabet, and doing so in time proportional to the number of arrays generated. A previously published algorithm that claims to solve this problem in constant time per array generated is shown to be incorrect, and new algorithms are proposed. We state as open the design of an equally efficient on-line algorithm for this problem.

1 Introduction

The classical method for computing the border array $\boldsymbol{\beta} = \boldsymbol{\beta}[1..n]$ of a given string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is the so-called "failure function" algorithm [AHU74], that executes in $\Theta(n)$ time. A recent paper [MSM99] introduces the idea of *b-equivalent* strings — that is, strings with the same border array — and shows how to construct *b-canonical* strings that are the unique representatives of each b-equivalent class. The paper then describes an algorithm to generate all possible border arrays of length n together with their corresponding b-canonical strings in time

*communicating author (smyth@mcmaster.ca); also at School of Computing, Curtin University, Perth WA 6845, Australia.

proportional to the number of arrays generated. If b_n denotes the number of distinct border arrays of length n, the sequence

$$\begin{aligned} B &= \{b_1, b_2, \dots.\} \\ &= \{1, 2, 4, 9, 20, 47, 110, 263, 630, 1525, \dots.\} \end{aligned}$$

is shown to be a new integer sequence [SP95].

In this paper we extend the results of [MSM99] in two ways:

(1) We describe a $\Theta(n)$-time algorithm that determines whether or not a given array $\boldsymbol{y}[1..n]$ of integers is a border array of some string (on an unbounded alphabet).

(2) We show that the [MSM99] algorithm to generate all possible border arrays is actually incorrect, in the sense that it requires more than constant time per string generated. We then describe a time- and space-optimal algorithm that generates all border arrays of length at most n (on a bounded or unbounded alphabet) without the need to store the underlying b-canonical strings. These arrays constitute a new infinite class of integer sequences.

We state as open problems the design of a modified algorithm (1) that executes on an alphabet of bounded size α while achieving the same time complexity as the original, and the design of an on-line algorithm (2) that achieves the same time complexity as the original.

2 Identifying Valid Border Arrays

This paper deals with arrays $\boldsymbol{y} = \boldsymbol{y}[1..n]$ of nonnegative integers. For these arrays it will be convenient to make use of the notation $\boldsymbol{y}^1[i] = \boldsymbol{y}[i]$ for every $i \in 1..n$, while

$$\boldsymbol{y}^j[i] = \boldsymbol{y}\big[\boldsymbol{y}^{j-1}[i]\big]$$

for every $j > 1$ such that $\boldsymbol{y}^{j-1}[i] \in 1..n$. It follows from the definition of border that for a border array $\boldsymbol{\beta}$, $0 \le \boldsymbol{\beta}[i] < i$ for every i, so that the sequence $i, \boldsymbol{\beta}[i], \boldsymbol{\beta}^2[i], \dots$ is monotone decreasing to zero, hence finite. We state a well-known result [AHU74]:

Lemma 2.1 *For some integer $n \ge 1$, let $\boldsymbol{x} = \boldsymbol{x}[1..n]$ denote a string with border array $\boldsymbol{\beta}$. Let k be the integer such that $\boldsymbol{\beta}^k[n] = 0$. Then*

(a) for every integer $j \in 1..k$, $\boldsymbol{x}\big[1..\boldsymbol{\beta}^j[n]\big]$ is a border of $\boldsymbol{x}[1..n]$;

(b) for any choice of letter λ, every border of $\boldsymbol{x}[1..n{+}1] = \boldsymbol{x}[1..n]\lambda$ has a length that is an element of the following set:

$$\begin{aligned} S^n &= \{S_0^n, S_1^n, \dots, S_k^n\} \\ &= \big\{0, \boldsymbol{\beta}[n]{+}1, \boldsymbol{\beta}^2[n]{+}1, \dots, \boldsymbol{\beta}^k[n]{+}1\big\}. \quad \square \end{aligned}$$

The set S^n defined in Lemma 2.1(b) is called the *admissible set* of the border array $\boldsymbol{\beta}[1..n]$, and each of its elements S^n_j, $j = 0, 1, \ldots, k$ is called an *admissible extension* of $\boldsymbol{\beta}$. Thus the lemma tells us that the only possible border arrays $\boldsymbol{\beta}[1..n+1] = \boldsymbol{\beta}[1..n]m$ are those for which m is an admissible extension. To see that the converse is not true — that is, that not all admissible extensions give rise to border arrays — consider the following example ($n = 11$):

$$\begin{array}{rccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \boldsymbol{\beta} = & 0 & 0 & 1 & 1 & 2 & 3 & 2 & 3 & 4 & 5 & 6 \end{array}$$

Here the border array $\boldsymbol{\beta}$ corresponds to a string $\boldsymbol{x} = abaababaaba$, for example. In fact, it is easy to see that, up to an isomorphism on the alphabet, this string is the *only* one that corresponds to $\boldsymbol{\beta}$. ¿From Lemma 2.1(b) we see that the admissible extensions m of $\boldsymbol{\beta}$ are

$$m = \left\{ \begin{array}{lllllllll} 0 & & & & & & & & \\ \boldsymbol{\beta}[11]{+}1 & = & 7 & & & & & & \\ \boldsymbol{\beta}^2[11]{+}1 & = & \boldsymbol{\beta}[6]{+}1 & = & 4 & & & & \\ \boldsymbol{\beta}^3[11]{+}1 & = & \boldsymbol{\beta}^2[6]{+}1 & = & \boldsymbol{\beta}[3]{+}1 & = & 2 & & \\ \boldsymbol{\beta}^4[11]{+}1 & = & \boldsymbol{\beta}^3[6]{+}1 & = & \boldsymbol{\beta}^2[3]{+}1 & = & \boldsymbol{\beta}[1]{+}1 & = & 1 \end{array} \right.$$

Of these five admissible extensions, only three ($m = 0, 7, 4$) can actually be used to extend $\boldsymbol{\beta}$ to a border array $00112323456m$; these extensions correspond to appending the letters c, b, a, respectively, to $\boldsymbol{x}$. The extensions $m = 1, 2$ do not yield valid border arrays because, even though they give the lengths of borders of $\boldsymbol{x}[1..12] = \boldsymbol{x}[1..11]a$ and $\boldsymbol{x}[1..11]b$, respectively, they do not give the lengths of the *longest* borders.

In order to characterize those values of m that can be used to extend a border array $\boldsymbol{\beta}[1..n]$ to a border array $\boldsymbol{\beta}[1..n]m$, we make use of the following definition:

> A nonzero admissible extension m of a border array $\boldsymbol{\beta}$ is said to be *invalid* if and only if there exists an admissible extension m' of $\boldsymbol{\beta}$ such that $m = \boldsymbol{\beta}[m']$. Any other admissible extension of $\boldsymbol{\beta}$ is said to be a *valid* extension.

It follows from this definition that for $n \geq 1$ the admissible extensions

$$S^n_0 = 0, \quad S^n_1 = \boldsymbol{\beta}[n]{+}1$$

are always valid. We shall see in Theorem 2.2 that every valid extension determines a distinct border array; thus $b_n \geq 2^{n-1}$, as in fact we have seen in the sequence B whose first ten terms were given in the Introduction.

Observe that this definition implicitly assumes an unbounded alphabet. For example, the border array

$$\boldsymbol{\beta}[1..15] = 001012301234567$$

corresponds to a string

$$\boldsymbol{x} = abacabadabacaba$$

and has five valid extensions $m = 0, 8, 4, 2, 1$ that result from appending the letters e, d, c, b, a, respectively, to $\boldsymbol{x}$. However, if the alphabet size were limited to $\alpha = 4$, we would presumably not wish to regard $m = 0$ as "valid". The following theorem provides a justification for our use of this term.

Theorem 2.2 *For every $n \geq 1$, an integer array $\boldsymbol{y} = \boldsymbol{y}[1..n]$ is a border array if and only if $\boldsymbol{y}[1] = 0$ and each $\boldsymbol{y}[i]$ is a valid extension of $\boldsymbol{y}[1..i–1]$, $i = 2, 3, \ldots, n$.*

Proof The result is trivially true for $n = 1$, and so we may suppose $n \geq 2$.

To prove necessity, suppose that for some $i \in 1..n-1$,

$$\boldsymbol{y}[1..i] \quad \text{and} \quad \boldsymbol{y}[1..i+1] = \boldsymbol{y}[1..i]m$$

are both border arrays, and let $\boldsymbol{x} = \boldsymbol{x}[1..i+1]$ denote a string with border array $\boldsymbol{y}[1..i+1]$. By Lemma 2.1(b), m must be an admissible extension of $\boldsymbol{y}[1..i]$. We suppose however that m is invalid and derive a contradiction.

Since m is invalid, there exists an admissible extension $m' > m$ of $\boldsymbol{y}[1..i]$ such that $\boldsymbol{y}[m'] = m$. Then $m' = \boldsymbol{y}^r[i]+1$ for some integer $r \geq 1$, and the following statements are true:

(1) $\boldsymbol{x}[1..m] = \boldsymbol{x}[i-m+2..i+1]$ since $\boldsymbol{y}[i+1] = m$;

(2) $\boldsymbol{x}[1..m] = \boldsymbol{x}[m'-m+1..m']$ since $\boldsymbol{y}[m'] = m$;

(3) $\boldsymbol{x}[1..m'-1] = \boldsymbol{x}[i-m'+2..i]$ since $m'-1 = \boldsymbol{y}^r[i]$.

¿From (1) and (2) we conclude that

$$\boldsymbol{x}[m'] = \boldsymbol{x}[m] = \boldsymbol{x}[i+1],$$

so that (3) can be extended to

$$\boldsymbol{x}[1..m'] = \boldsymbol{x}[i-m'+2..i+1].$$

Thus $\boldsymbol{x}[1..i+1]$ has a border of length $m' > m$, contradicting the assumption that $\boldsymbol{y}[1..i+1] = \boldsymbol{y}[1..i]m$ is a border array. We conclude that m must be valid, as required.

To prove sufficiency, let $\boldsymbol{y} = \boldsymbol{y}[1..n]$ be an array such that $\boldsymbol{y}[1] = 0$ and each $\boldsymbol{y}[i]$ is a valid extension of $\boldsymbol{y}[1..i-1]$, $i = 2, 3, \ldots, n$. We show by induction that $\boldsymbol{y}$ is a border array of some string.

Since $\boldsymbol{y}[1] = 0$, the result holds for $n = 1$. Suppose then that for $n \geq 2$ and some $i \in 2..n$, $\boldsymbol{y} = \boldsymbol{y}[1..i-1]$ is a border array of some string $\boldsymbol{x}[1..i-1]$. We show that therefore $\boldsymbol{y}[1..i]$ must be a border array.

Let $m = \boldsymbol{y}[i]$. By hypothesis m is a valid extension of $\boldsymbol{y}[1..i-1]$ and so by Lemma 2.1(b) two cases arise:

$\underline{m = 0}$ In this case $\boldsymbol{y}[1..i]$ is a border array of a string $\boldsymbol{x}[1..i-1]\lambda$, where the letter λ is chosen to be distinct from every previous letter in $\boldsymbol{x}[1..i-1]$.

$\underline{m > 0}$ Here $m = \boldsymbol{y}^p[i-1]+1$ for some integer $p \geq 1$, so that by the inductive hypothesis $\boldsymbol{x}[1..m-1]$ is a border of $\boldsymbol{x}[1..i-1]$. Then we can choose $\boldsymbol{x}[i] = \boldsymbol{x}[m]$, so that $\boldsymbol{x}[1..m]$ is a border of $\boldsymbol{x}[1..i]$ — we want to show that it is the longest border.

If $\boldsymbol{x}[1..m]$ is not the longest border of $\boldsymbol{x}[1..i]$, there must exist a longer border $\boldsymbol{x}[1..m']$ such that $m = \boldsymbol{y}[m']$. By Lemma 2.1(b), $m' = \boldsymbol{y}^r[i-1]+1$ for some positive integer $r < p$. But then by definition m is invalid, contrary to the original assumption that each $\boldsymbol{y}[i]$ is a valid extension of $\boldsymbol{y}[1..i-1]$. We conclude that $\boldsymbol{y}[1..i]$ is the border array of the string $\boldsymbol{x}[1..i] = \boldsymbol{x}[1..i-1]\boldsymbol{x}[m]$, as required.

□

This theorem makes clear that an extension $m = \boldsymbol{y}[i]$ of a border array $\boldsymbol{y} = \boldsymbol{y}[1..i-1]$ yields a border array $\boldsymbol{y}[1..i]$ if and only if

(1) m is an admissible extension of $\boldsymbol{y}$;

(2) there exists no admissible extension $m' > m$ of $\boldsymbol{y}$ such that $\boldsymbol{y}[m'] = m$.

The algorithm that determines whether or not a given array is a border array simply evaluates these two conditions in a straightforward manner for every position $i \in 2..n$. Thus the outline of the algorithm can be expressed as follows:

```
  — For y[1..n], n ≥ 1, return either n+1
  — or the first position i ∈ 1..n
  — such that y[i] is invalid.
if y[1] ≠ 0 then return 1
else
  i ← 2
  while i ≤ n and also valid(i, y[1..i]) do
    i ← i+1
  return i
```

The Boolean function *valid* returns `TRUE` if and only if conditions (1) and (2) are satisfied by $m = \boldsymbol{y}[i]$, as shown in Figure 1. This function assumes that the admissible extensions S_j^{i-1} are given in the order shown in Lemma 2.1(b), so that the final one is $S_k^{i-1} = 1$. Observe that the algorithm described here makes no reference to any corresponding string $\boldsymbol{x}$, but bases its determination of validity entirely on the properties of the given array $\boldsymbol{y}$.

Thus, based on Theorem 2.2 and this discussion, we may conclude that our algorithm is correct. To see that it executes in $\Theta(n)$ time, we need to show that the total number of operations performed in the **repeat** and **while** loops of function *valid* is $O(n)$. But this fact follows from the corresponding result for the failure function algorithm [AHU74]: in that algorithm the border array

```
function valid(i, y[1..i])
  — Given that y[1..i−1] is a border array,
  — return TRUE iff y[i] is valid.

  — First determine whether y[i] is admissible.
j ← −1
repeat j ← j+1
until y[i] = S_j^{i−1} or S_j^{i−1} = 1
if y[i] ≠ S_j^{i−1} then return FALSE
else

  — Next determine whether y[i] satisfies condition (2).
  j′ ← 1
  while j′ < j and y[i] ≠ y[S_{j′}^{i−1}] do
    j′ ← j′+1
  return (j′ ≥ j)
```

Figure 1: The Boolean Function *valid*

elements $\boldsymbol{\beta}^j[i-1]$, $j = 1, 2, \ldots, k$, are inspected just as they are in *each of* the **repeat** and **while** loops. In the failure function algorithm the total number of inspections is at most n because each inspection reduces the possible length of the longest border by at least one. In the present algorithm the possible number of inspections is at most doubled to $2n$ because essentially the same inspections are carried out in each of two loops: first j is incremented by steps of one to its correct value, while at most j values are tested in the **while** loop. So the result is still a linear time algorithm, a fact we state formally in the second main result of this section:

Theorem 2.3 *The algorithm presented in this section correctly determines in time $\Theta(n)$ whether or not a given integer array $\boldsymbol{y}[1..n]$ is a border array.* □

To conclude this section, we remark that a version of Theorem 2.2 appears as Theorem 3.2 in [MSM99]; however, the result as it is given there is much less clear and its proof depends on an elaborate theory of b-canonical strings that we have avoided here with a proof that is elementary. We remark also that while Theorem 2.2 can be modified in an obvious way to hold for an alphabet of bounded size α, corresponding modification to the algorithm does not seem to be so straightfoward. The design of the algorithm for a bounded alphabet is left as an open problem.

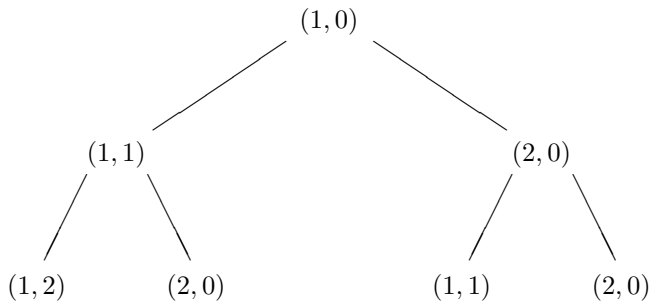

Figure 2: Trie T_3 — All Border Arrays of Length $k \leq 3$

3 Computing All Border Arrays of Length At Most n

In [MSM99] all the border arrays of length at most n are generated by growing a trie T_n of height n in which every simple path of length $k \leq n$ from the root spells out both a unique border array $\boldsymbol{\beta} = \boldsymbol{\beta}[1..k]$ and the b-canonical string $\boldsymbol{x}[1..k]$ corresponding to $\boldsymbol{\beta}$. Thus each node of T_n may be thought of as being labelled with an integer pair (i, β), where i denotes the i^{th} smallest letter λ_i in an ordered standard alphabet, and β is the value of a corresponding entry in the border array $\boldsymbol{\beta}$. For $n = 3$, T_n appears as shown in Figure 2, representing strings

$$\lambda_1\lambda_1\lambda_1,\ \lambda_1\lambda_1\lambda_2,\ \lambda_1\lambda_2\lambda_1,\ \lambda_1\lambda_2\lambda_2$$

with corresponding border arrays

$$012, 010, 001, 000.$$

The algorithm described in [MSM99] uses the canonical string $\boldsymbol{x}[1..k]$ spelled out by the path from the root to the current node N as a means of determining the children of N. Effectively, standard letters $\lambda_1, \lambda_2, \ldots$ are appended one-by-one to $\boldsymbol{x}[1..k]$ yielding new strings $\boldsymbol{x}[1..k]\lambda_i$, $i = 1, 2, \ldots$; for each new string formed, the corresponding $(k+1)^{\text{th}}$ border array element is computed. The process terminates when a standard letter, say λ_r, is appended for which the corresponding border array value is zero (see Lemma 3.4 in [MSM99]). Thus for each new node of T_n, one step in the failure function calculation is performed, requiring amortized constant time as discussed in Section 2. Hence the claim that T_n is constructed in time proportional to the number of nodes; that is, proportional to the number of border arrays (and corresponding strings) generated.

But the algorithm described in [MSM99] generates T_n in a breadth-first or on-line manner: T_{k+1} is actually computed from the leaf nodes of T_k for

every $k \in 1..n-1$. Since for every node N both the corresponding $\boldsymbol{x}[1..k]$ and $\boldsymbol{\beta}[1..k]$ need to be available for the failure function calculation, $\Theta(k)$ time will be required to traverse the path from the root to node N in order to compute them. Thus the time required to compute each child of node N in a breadth-first algorithm is not constant, but rather $\Theta(k/r)$, where r is the number of children of N.

The obvious correction to the [MSM99] algorithm is to build T_n in a depth-first manner that uses two working-storage arrays $\boldsymbol{x} = \boldsymbol{x}[1..n]$ and $\boldsymbol{\beta}[1..n]$ to store the path from the root of T_n to the current node N. Then for each node N, the current values $\boldsymbol{x}[1..k]$ and $\boldsymbol{\beta}[1..k]$ are known and can be used to compute the children of N: each extension $\boldsymbol{x}[1..k]\lambda_i$, $i = 1, 2, \ldots, r$, can be formed so that corresponding extensions $\boldsymbol{\beta}[1..k]\lambda_i$ can be computed using a single constant-time step of the failure function algorithm. A depth-first recursion that computes all the children of each N before computing any of N's siblings will then lead to the result claimed in [MSM99]: T_n will be constructed in $\Theta(b_n)$ time using $\Theta(b_n)$ space.

The depth-first approach also enables us to solve efficiently a problem raised in [MSM99]: the computation of a trie T'_n whose node labels consist only of border array values β, omitting the elements of the underlying canonical string $\boldsymbol{x}$. This can easily be accomplished by maintaining the working storage array $\boldsymbol{x}[1..n]$ but not storing the current letter in the current node N: the algorithm will execute recursively in exactly the same way.

Note that it is straightforward to modify each of these depth-first algorithms to generate a trie for a given bounded alphabet A of size α: it is necessary only to replace the number m of children computed at each node by $\min\{r, \alpha\}$. We can now state formally the main result of this section:

Theorem 3.1 *For any given positive integer n, the two algorithms outlined in this section compute all possible border arrays of length at most n on either a bounded or unbounded alphabet in time $\Theta(b_n)$ and space $\Theta(b_n)$, where b_n is the number of arrays generated.* □

We remark that the depth-first algorithms described above have the disadvantage that they provide no means of efficiently computing T_{n+1} (respectively, T'_{n+1}) from T_n (respectively, T'_n). We leave as an open problem the design of on-line (breadth-first) algorithms that perform the same computation with equal efficiency.

References

[AHU74] Alfred V. Aho, John E. Hopcroft & Alfred D. Ullman, *The Design & Analysis of Computer Algorithms*, Addison-Wesley (1974).

[MSM99] Dennis Moore, W. F. Smyth & Dianne Miller, **Counting distinct strings**, *Algorithmica 23* (1999) 1-13.

[SP95] N. J. A. Sloane & Simon Plouffe, *The Encyclopedia of Integer Sequences*, Academic Press (1995). See also

`http://www.research.att.com/~njas/sequences/`

Acknowledgements

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