Verifying a Border Array in Linear Time*

František Franěk[1], Shudi Gao[1], Weilin Lu[1], P. J. Ryan[1], W. F. Smyth[1,2], Yu Sun[1], and Lu Yang[1]

[1] Algorithms Research Group, Department of Computing & Software
McMaster University, Hamilton, Ontario, Canada L8S 4L7
smyth@mcmaster.ca
www.cas.mcmaster.ca/cas/research/groups.shtml

[2] School of Computing, Curtin University, GPO Box U-1987
Perth WA 6845, Australia

Abstract. A *border* of a string x is a proper (but possibly empty) prefix of x that is also a suffix of x. The *border array* $\beta = \beta[1..n]$ of a string $x = x[1..n]$ is an array of nonnegative integers in which each element $\beta[i]$, $1 \le i \le n$, is the length of the longest border of $x[1..i]$. In this paper we first present a simple linear-time algorithm to determine whether or not a given array $y = y[1..n]$ of integers is a border array of some string on an alphabet of unbounded size and then a slightly more complex linear-time algorithm for an alphabet of any given (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 conclude with an equally efficient on-line algorithm for this problem.

1 Introduction

The classical method for computing the border array $\beta = \beta[1..n]$ of a given string $x = x[1..n]$ is the so-called "failure function" algorithm [1], that executes in $O(n)$ time. A recent paper [3] introduces the idea of *b-equivalent* — that is, strings with the same border array — and shows how to construct, on a standard alphabet, *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 proportional to the number of arrays generated. If b_n denotes the number of distinct border arrays of length n that can exist when the alphabet

* Supported in part by grants from the Natural Sciences & Engineering Research Council of Canada.

is unbounded, then the sequence

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

is shown to be a new integer sequence [4].

In this paper we extend the results of [3] in two ways:

(1) We describe an $O(n)$-time algorithm that determines whether or not a given array $y[1..n]$ of integers is a border array of some string (on a bounded or unbounded alphabet)
(2) We show that the algorithm [3] 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.

    Unlike the algorithm described in [3], this algorithm generates a trie in a depth-first fashion and so is not *on-line* — that is, the set of border arrays for $n+1$ cannot be efficiently derived from the set of border arrays for n. We conclude by describing another algorithm that is on-line and that achieves constant time per string generated.

2 Identifying Valid Border Arrays For Unbounded Alphabets

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

$$y^j[i] = y\left[y^{j-1}[i]\right]$$

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

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

(a) for every integer $j \in 1..k$, $x\left[1..\beta^j[n]\right]$ is a border of $x[1..n]$;
(b) for any choice of letter λ, every border of $x[1..n+1] = x[1..n]\lambda$ has a length that is an element of the following set:

$$\begin{aligned} S^n &= \{S^n_0, S^n_1, \ldots, S^n_k\} \\ &= \left\{0, \beta[n]+1, \beta^2[n]+1, \ldots, \beta^k[n]+1\right\}. \quad \square \end{aligned}$$

The set S^n defined in Lemma 1(b) is called the ***admissible set*** of the border array $\beta[1..n]$, and each of its elements S_j^n, $j = 0, 1, \ldots, k$ is called an ***admissible extension*** of β. Thus the lemma tells us that the only possible border arrays $\beta[1..n+1] = \beta[1..n]m$ 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 ($n = 11$):

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

Here the border array β corresponds to a string $x = abaababaaba$, 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 β. From Lemma 1(b) we see that the admissible extensions m of β are

$$m = \begin{cases} 0 \\ \beta[11]+1 \ = 7 \\ \beta^2[11]+1 = \beta[6]+1 \ = 4 \\ \beta^3[11]+1 = \beta^2[6]+1 = \beta[3]+1 \ = 2 \\ \beta^4[11]+1 = \beta^3[6]+1 = \beta^2[3]+1 = \beta[1]+1 = 1 \end{cases}$$

Of these five admissible extensions, only three ($m = 0, 7, 4$) can actually be used to extend β to a border array $00112323456m$; these extensions correspond to appending the letters c, b, a, respectively, to x. (Of course, in place of c, any letter other than a or b could be used.) The extensions $m = 1, 2$ do not yield valid border arrays because, even though they give the lengths of borders of $x[1..12] = x[1..11]a$ and $x[1..11]b$, 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 $\beta[1..n]$ to a border array $\beta[1..n]m$, we make use of the following definition:

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

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

$$S_0^n = 0, \quad S_1^n = \beta[n]+1$$

are always valid. We shall see in Theorem 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 is useful only for an unbounded alphabet. For example, the border array

$$\beta[1..15] = 001012301234567$$

corresponds to a string

$$x = abacabadabacaba$$

and has five valid extensions $m = 0, 8, 4, 2, 1$ that result from appending the letters e, d, c, b, a, respectively, to 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 *For every $n \geq 1$, an integer array $y = y[1..n]$ is a border array if and only if $y[1] = 0$ and each $y[i]$ is a valid extension of $y[1..i-1]$, $i = 2, 3, \ldots, n$.*

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

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

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

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

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

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

We conclude from (1) and (2) that

$$x[m'] = x[m] = x[i+1],$$

so that (3) can be extended to

$$x[1..m'] = x[i-m'+2..i+1].$$

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

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

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

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

$\underline{m = 0}$

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

$\underline{m > 0}$

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

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

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

(1) m is an admissible extension of y;
(2) there exists no admissible extension $m' > m$ of y such that $y[m'] = m$.

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 main algorithm can be expressed as follows:

```
  — For $y[1..n]$, $n \geq 1$, return either $n{+}1$
  — or the first position $i \in 1..n$
  — such that $y[i]$ is invalid.
if $y[1] \neq 0$ then return 1
  — repeatedly call the function valid to check each
  — value $y[i]$ until the whole array $y$ is processed
$i \leftarrow 2$
while $i \leq n$ and $valid(i, y[1..i])$ do
  $i \leftarrow i{+}1$
return $i$
```

The Boolean function *valid* returns TRUE if and only if conditions (1) and (2) are satisfied by $m = y[i]$, as shown in Figure 1. Since the algorithm processes y position-by-position from left to right, terminating whenever an invalid position is found, we may assume that for every $i \geq 2$, $y[1..i-1]$ is a valid 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.
if $y[i] = 0$ then return TRUE
else
  $b \leftarrow y[i-1]$
  while $b > 0$ and $y[i] \neq b+1$ do
    $b \leftarrow y[b]$                          (1.1)
  if $y[i] \neq b+1$ then return FALSE
  else
  — Now determine whether $y[i] = b+1$ satisfies condition (2).
    $b' \leftarrow y[i-1]$
  while $b' > b$ and $b+1 \neq y[b'+1]$ do
      $b' \leftarrow y[b']$                        (1.2)
    return $(b' \leq b)$
```

Fig. 1. The Boolean Function *valid*

Observe that the algorithm described here makes no reference to any corresponding string x, but bases its determination of validity entirely on the properties of the given array y.

Thus, based on Theorem 2 and this discussion, we may conclude that our algorithm is correct. To see that it executes in $O(n)$ time, we need to show that the total number of operations performed in the **while** loops of function *valid* is $O(n)$. But this follows as in the failure function algorithm

The same analysis applies to the case where FALSE is returned except that the algorithm may terminate earlier, thus executing fewer steps. We conclude that our algorithm executes in linear time. This establishes the second main result of this section:

Theorem 3 *The algorithm presented in this section correctly determines in time $O(n)$ whether or not a given integer array $y[1..n]$ is a border array.* □

To conclude this section, we remark that a version of Theorem 2 appears as Theorem 3.2 in [3]; 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.

3 Identifying Valid Border Arrays For Bounded Alphabets

As shown in the previous section, the definition of a *valid extension* of a border array $y[1..n]$ is not appropriate for dealing with strings over an alphabet of a fixed finite size α. The example there shows that the problem is how to determine when 0 is valid in relation to the size of the alphabet. For the rest of this section we assume that $\alpha \geq 2$ is the fixed finite alphabet size. We capture the revised notion of validity in the following definition:

> A nonzero admissible extension m of a border array $\beta = \beta[1..n]$ is said to be an *α-valid extension of β* if and only if it is a valid extension of β (as defined in the previous section). 0 is said to be an *α-valid extension of β* if and only if for the set
>
> $$M_n = \{m : m > 0, m \ an \ \alpha - valid \ extension \ of \ \beta\},$$
>
> $|M_n| < \alpha$.

Lemma 4 below shows that $|M_n|$ is precisely the number of letters (of any alphabet of size α) used in all possible extensions of all previous borders in any string of which β is a border array.

Let us illustrate using the example of the previous section, where 8,4,2, and 1 are all valid (hence α-valid) extensions of the border array $\beta[1..15] =$ 001012301234567. The string $x = abacabadabacaba$ is a string of which β is a border array, and so we see that the alphabet $\{a, b, c, d\}$ must have size at least 4. Hence if $\alpha = 4$, 0 is not α-valid, while if $\alpha \geq 5$, it is α-valid (appending e to x gives 0).

Lemma 4 *Let $n \geq 2$ be an integer, and let $y[1..n]$ be a border array. Let m_1 and $m_2 \neq m_1$ be two distinct α-valid extensions of $y[1..n]$. Then for any string $x[1..n]$ such that $y[1..n]$ is its border array, $x[m_1] \neq x[m_2]$.*

Proof. By contradiction. We assume that there exists a string x such that $x[m_1] = x[m_2]$, where $m_1 = y^i[n]+1$ and $m_2 = y^j[n]+1$ for some $1 \leq i < j \leq k$ (k being the smallest integer such that $y^k[n] = 0$). Moreover assume that, for given j, the integer i is the maximum value satisfying this condition.

Since $x[y^i[n]+1] = x[y^j[n]+1]$ and $i < j$, $y[y^i[n]+1] \geq y^j[n]+1$. Since $y^j[n]+1$ is α-valid, $y[y^i[n]+1] \neq y^j[n]+1$, and so $y[y^i[n]+1] > y^j[n]+1$. Furthermore, $y[y^i[n]+1] = y^r[n]+1$ for some r, $i < r \leq k$. Thus

$$x[y^r[n]+1] = x[y^i[n]+1] = x[y^j[n]+1].$$

Since $y^r[n]+1 > y^j[n]+1$, $1 \leq i < r < j \leq k$. But this contradicts the maximality of i. □

The following theorem is an obvious modification of Theorem 2 for bounded alphabets.

Theorem 5 *For every $n \geq 1$, and for any $\alpha \geq 2$, an integer array $y = y[1..n]$ is a border array of a string over an alphabet of size α if and only if $y[1] = 0$ and each $y[i]$ is an α-valid extension of $y[1..i-1]$, $i = 2, 3, \ldots, n$.*

Proof. If $y[i] > 0$, the proof follows the same argument as in the proof of Theorem 2. Below, we cover the case when $y[i] = 0$.

Necessity: let $y[1..i]$ be a border array and $y[i] = 0$. By contradiction assume that 0 is not an α-valid extension of $y[1..i-1]$; that is, that $|M_{i-1}| = \alpha$. Take any string $x[1..i]$ of which $y[1..i]$ is a border array. Then for some α-valid $m > 0$, $x[m] = x[i]$. It follows that $y[i] \geq m > 0$, a contradiction.

Sufficiency: let $y[1..i-1]$ be a border array and suppose that $|M_{i-1}| < \alpha$. Take any string $x[1..i-1]$ of which $y[1..i-1]$ is a border array. Suppose $\lambda \neq x[m]$ for any $m \in M_{i-1}$ (The fact that such a λ exists follows from Lemma 4.) We will show that $y[1..i-1]0$ is a border array of $x[1..i-1]\lambda$. By contradiction assume that the border array $y[1..i]$ of $x[1..i-1]\lambda$ is such that $y[i] \neq 0$. But then by this theorem for nonzero values, $y[i] = m$ for some α-valid m, hence $\lambda = x[i] = x[m]$, a contradiction. □

This result makes it clear that for $y[i] \neq 0$, the algorithm of Section 2 can be used as it stands to determine whether or not $y[i]$ is α-valid. However, whenever $y[i] = 0$ and $i > 1$, the definition of α-valid requires additional processing to determine whether or not the number of valid nonzero extensions of $y[1..i-1]$ equals α. This processing can be viewed as a modification to function *valid*, that affects only the case $y[i] = 0$. The code is presented in Figure 2. A Boolean array $V[1..n]$ indicates whether or not each admissible extension m is valid ($V[m]$ = TRUE) or not valid ($V[m]$ = FALSE). We initialize $V[1..n]$ to FALSE before the first invocation of *valid*.

The new code is straightforward and falls into four parts:

* determine the number k of nonzero admissible extensions m and set each corresponding $V[m]$ to TRUE;
* identify invalid admissible extensions and thus compute v, the number of nonzero valid admissible extensions;
* reset to FALSE the components of V that were changed in the first part;
* return TRUE if and only if $v < \alpha$.

We claim that the revised algorithm correctly determines whether or not a given array y is a border array of some string on an alphabet of size α.

To analyze the complexity of the revised algorithm, we need to show that the extra processing involved when $y[i] = 0$ still requires only $O(n)$ steps. To

```
if $y[i] = 0$ then
  $k \leftarrow 0$; $b \leftarrow i - 1$
  while $b \neq 0$ do
    $k \leftarrow k+1$; $b \leftarrow y[b]$; $V[b + 1] \leftarrow$ TRUE                (2.1)
  $v \leftarrow k$; $b \leftarrow i - 1$
  while $b \neq 0$ do
    $b \leftarrow y[b]$                        (2.2)
    $b' \leftarrow b + 1$
    if $V[b']$ then
      while $y[b'] > 0$ do
        $v \leftarrow v-1$; $b' \leftarrow y[b']$; $V[b'] \leftarrow$ FALSE          (2.3)
  $b \leftarrow i - 1$
  while $b \neq 0$ do
    $b \leftarrow y[b]$; $V[b + 1] \leftarrow$ FALSE
  return $(v < \alpha)$

else

   — (The rest of function valid follows.)
```

Fig. 2. Determining whether $y[i] = 0$ is α-valid or not

this end, fix such an i and let s be the number of times (2.1) is executed. Then $s \leq y[i - 1] + 1$ since b is set to $y[i - 1]$ on the first pass and is reduced by at least 1 on each subsequent pass. Note that s distinct components of V are set to TRUE.

Now the second **while** loop is controlled by the same parameters as the first. In particular, (2,2) is executed s times. Further, each execution of (2.3) sets to FALSE a component of V that was previously TRUE. Thus (2.3) executes at most s times in total — more than once on some passes and not at all on others. The third **while** loop (which resets V) is also executed s times.

Now, as in Section 2, $b = y[i - 1]$ increases by at most 1 on each call of *valid*. However, b decreases by $y[i - 1]$ when $y[i] = 0$. Thus the sum of all these $y[i - 1]$ cannot exceed $n - 1$ and hence the sum of the corresponding values of s cannot exceed $2(n - 2)$. It follows that overall, $O(n)$ time is sufficient to handle the cases in which $y[i] = 0$, in spite of the nested **while** loops. Since the remainder of the processing is unchanged, we conclude:

Theorem 6 *The algorithm presented in this section correctly determines in time $O(n)$ whether or not a given integer array $y[1..n]$ is a border array on an alphabet of specified size α.* □

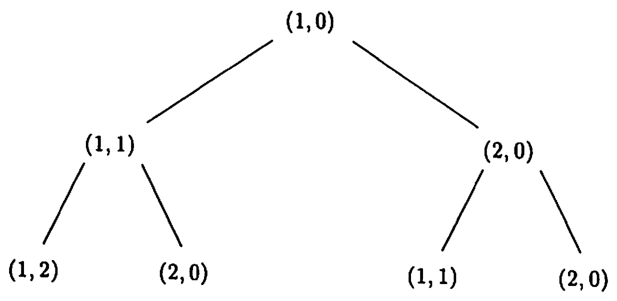

Fig. 3. Trie T_3 — All Border Arrays of Length $k \leq 3$

4 Computing All Border Arrays of Length At Most n

In [3] 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 $\beta = \beta[1..k]$ and the b-canonical string $x[1..k]$ corresponding to β. 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 β. For $n = 3$, T_n appears as shown in Figure 3, 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 [3] uses the canonical string $x[1..k]$ 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 $x[1..k]$ yielding new strings $x[1..k]\lambda_i$, $i = 1, 2, \ldots$; for each new string formed, the corresponding $(k{+}1)^{\text{st}}$ 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 [3]). 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 [3] 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 $x[1..k]$ and $\beta[1..k]$ 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 [3] algorithm is to build T_n in a depth-first manner that uses two working-storage arrays $x = x[1..n]$ and $\beta[1..n]$ to store the path from the root of T_n to the current node N. Then for each node N, the current values $x[1..k]$ and $\beta[1..k]$ are known and can be used to compute the children of N: each extension $x[1..k]\lambda_i$, $i = 1, 2, \ldots, r$, can be formed so that corresponding extensions of $\beta[1..k]$ 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 [3]: 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 [3]: 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 x. This can easily be accomplished by maintaining the working storage array $x[1..n]$ 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 7 *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). In the next section, we provide an on-line (breadth-first) algorithm that performs the same computation with equal efficiency.

5 An On-line Algorithm for Computing All Border Arrays of Length At Most n

We wish to construct the trie T_n in a breadth-first fashion using only constant time to compute the border array entry of each node. The failure function algorithm satisfies this requirement, but it needs the corresponding arrays x and β to be available. The elements of the two arrays are already stored in the ancestor nodes of the tree. We will modify the failure function algorithm to use this information instead of the two arrays.

The failure function algorithm accesses the arrays x and β in the following two ways:

(a) $\beta[b]$, which returns the length of the longest border of the position b; and
(b) $x[b+1]$, which returns the letter at the next position of b.

Each node in our trie T_n will represent a border array/b-canonical string pair. Then the above operation (a) is equivalent to finding the node corresponding to the longest border of the string represented by the current node. Operation (b) is equivalent to finding one of the children of the current node. We refine the tree structure of T_n to simulate these two operations.

First we introduce an extra node E which corresponds to the 0-position or empty string/border. A pointer in E points to the "real" root R of T_n: $(\lambda_1, 0, 1)$, which means that the current letter is λ_1, with empty border, and the length of the string is 1.

Except for E, each node N in T_n represents a border array $\beta[1..j]$ and its b-canonical string $x[1..j]$. We have the following data in each node N:

λ	the letter $x[j]$;
β	the border array entry $\beta[j]$;
j	the length of the corresponding border array/b-canonical string;
pr	a pointer to the parent: the node of $\beta[1..j-1]/x[1..j-1]$;
br	a pointer to the border: the node of $\beta[1..\beta[j]]/x[1..\beta[j]]$;
ne	a pointer to the next node of the same level as N;
cp	a list of pointers to the children of N;
ci	a pointer to the child whose descendant is currently being processed.

The pointers *ci* are maintained to form a path from E to the node N to which a child N' is being added. Note that j is just the level in the tree where $E.j = 0$. Also, we define $E.ci = R$ and $R.br = E$. These values do not change.

We now describe the construction of T_n. Beginning with E, each new level is an ordered sequence of nodes. Successive children of a given node are constructed by appending a different letter (using the ordering of the alphabet) to the string represented by the parent until an empty border results. The algorithm for computing the border is described below.

When adding a new node, it is trivial to determine the values of λ, j, *pr*, *ne* and *cp*. Also, β is just the level pointed to by *br*. We now show how to compute *br* and *ci*:

br: When adding a child N' to the current node N, we proceed as follows:

```
$N_b \leftarrow N.br; N_{b+1} \leftarrow N_b.ci$
while $N_b \neq E$ and $N'.\lambda \neq N_{b+1}.\lambda$ do
    $N_b \leftarrow N_b.br; N_{b+1} \leftarrow N_b.ci$
if $N'.\lambda = N_{b+1}.\lambda$ then
    $N'.br \leftarrow N_{b+1}$
else
    $N'.br \leftarrow E$
$N'.\beta \leftarrow N'.br.j$
```

First we use *N.br* to find the border node N_b of N, then use $N_b.ci$ to find the next node N_{b+1} along the path from N_b to N. Now we compare the letters of N' and N_{b+1}. We continue to do so until we find a match or reach the empty border. Finally we set the *br* and β according to whether the letters match or not.

ci: When a first child of N is added, we can set *N.ci* to this child. This value does not have to be changed until we start adding grandchildren for N. When finished adding children to a node N, we use *N.ne* to find the next node. Now we need to update the *ci* of ancestor nodes of N. For example, suppose that a node N_p has two children N_1 and N_2. When we are adding children to N_1, $N_p.ci = N_1$. Now we want to add children to N_2. We need to update *ci* so that $N_p = N_2$. Each time we finish adding children to a node N, we call the procedure *update_ci*(*N.pr*). It is possible that N is the last child of its parent N_p. Then we may need to update the *ci* of the parent nodes recursively. In the following, *next* denotes the pointer field in the linked list *cp*, and *cp.first* returns the first node of *cp*:

```
update_ci(N.pr)
procedure update_ci($N_p$)
    if ci.next $\neq$ NULL then
        ci $\leftarrow$ ci.next
    else
        ci $\leftarrow$ cp.first
        if $N_p \neq R$ then
            update_ci($N_p$.pr)
```

We update *ci* to point to the next child, if there is any. Otherwise, we set *ci* to point to the first child, and update *ci* of the parent node.

According to the above description, we have an on-line algorithm that computes all the border arrays/*b*-canonical strings of length n. Except for *cp*, all data in each node require constant space. The total number of nodes in *cp* is equal to the total number of child nodes in the tree, which is $|T_n| - 1$. So the algorithm still needs space proportional to the number of arrays generated.

The code for *br* is basically the same as the failure function algorithm. Thus it takes amortized constant time for each node. When we generate T_{n+1} from T_n, the number of times we update *ci* is the same as the number of nodes in *cp*, which is $|T_n| - 1$. Thus we get the following conclusion about the complexity of the algorithm:

Theorem 8 *For every positive integer n, the algorithm outlined in this section computes all border arrays (b-canonical strings) of length n in $\Theta(b_n)$ time and represents them in $\Theta(b_n)$ space.* □

An easy modification can be made to the algorithm for a bounded alphabet of size $\alpha \geq 2$. Theorem 8 also holds in this case.

References

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

[2] Donald E. Knuth, James H. Morris & Vaughan R. Pratt, **Fast pattern matching in strings**, *SIAM J. Comput. 6-2* (1977) 323-350.

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

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

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