Efficient validation and construction of border arrays*

Jean-Pierre Duval Thierry Lecroq Arnaud Lefebvre

LITIS, University of Rouen, France,
{Jean-Pierre.Duval,Thierry.Lecroq,Arnaud.Lefebvre}@univ-rouen.fr

Abstract

In this article we present an on-line linear time and space algorithm to check if an integer array f is the border array of at least one string w built on a bounded or unbounded size alphabet Σ. We first show some relations between the border array of some string w and the skeleton of the DFA recognizing $\Sigma^* \cdot w$, independently of the explicit knowledge of w. This enables us to design algorithms for validating and generating border arrays that outperform existing ones [4, 3]. The validating algorithm lowers the delay (time spent on one element of the array) from $O(|w|)$ to $O(\min\{|\Sigma|, |w|\})$ comparing to algorithms in [4, 3]. Finally we give some results on the numbers of distinct border arrays on some alphabet sizes.

1 Introduction

A border u of a string w is a prefix and a suffix of w such that $u \neq w$. The computation of the border array of a string w i.e. of the borders of each prefix of a string w is strongly related to the string matching problem: given a string w, find the first or, more generally, all its occurrences in a longest string y. The border array of w is better known as the "failure function" introduced in [8] (see also [1]). In [4] (see also [11]) a method is presented to check if an integer array f is a border array for some string w. The authors first give an on-line linear time algorithm to verify if f is a border array on an unbounded size alphabet. Then they give a more complex algorithm that works on a bounded size alphabet. In [3] a simpler algorithm is presented for this case. Furthermore if f is a border array we were able to build, on-line and in linear time, a string w on a minimal size alphabet for which f is the border array. The resulting algorithm is elegant and integrates three parts: the checking on an unbounded alphabet, the checking on a bounded size alphabet and the design of the corresponding string if f is a border array. The first two parts can work independently. In the present article we give a more elegant presentation of this result. Moreover we present new results concerning the relation between the border array f and the skeleton of the deterministic finite automaton recognizing $\Sigma^* \cdot w$. Actually these results are completely independent from w. We then present a new linear time

*This work was partially supported by the project "Algorithmique génomique" of the program "MathStic" of the french CNRS.

and space on-line algorithm that checks if a given integer array is a border array of some string. This algorithm lowers the delay (time spent on one element of the array) from $O(|w|)$ to $O(\min\{|\Sigma|, |w|\})$ comparing to algorithms in [4, 3]. An easy extension of this algorithm enables to generate all the distinct border arrays of some length in linear space and in time proportional to their number. This is useful for generating minimal test sets for various string algorithms. Finally, using this efficient construction algorithm, we count the number of distinct border arrays for some alphabet sizes. These last results extend those of [7].

The remaining of this article is organized as follows. The next section introduces basic notions and notations on strings. Section 3 recalls known results on the validation of border arrays. Section 4 presents our new results. Section 5 presents our new algorithm together with its correctness proof. In Sect. 6 we present results on the number of distinct border arrays. Finally we give our conclusions and perspectives in Sect. 7.

2 Notations and definitions

A *string* is a sequence of zero or more symbols from an alphabet Σ. The set of all strings over the alphabet Σ is denoted by Σ^*. We consider an alphabet of size s; for $1 \leq i \leq s$, $\sigma[i]$ denotes the i-th symbol of Σ. A string w of length n is represented by $w[1..n]$, where $w[i] \in \Sigma$ for $1 \leq i \leq n$. A string u is a *prefix* of w if $w = uv$ for $v \in \Sigma^*$. Similarly, u is a *suffix* of w if $w = vu$ for $v \in \Sigma^*$. A string u is a *border* of w if u is a prefix and a suffix of w and $u \neq w$. *The border* of a string w is the longest of its borders. It is denoted by $Border(w)$. The *border array* f of a string w of length n is defined by: $f[i] = |Border(w[1..i])|$ for $1 \leq i \leq n$ and we artificially set $f[0] = -1$. It is also known as the "failure function" of the Morris and Pratt string matching algorithm.

Example 1 The border array of `abababcaabcababa` is the following:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$w[i]$		a	b	a	b	a	c	a	a	b	c	a	b	a	b	a
$f[i]$	-1	0	0	1	2	3	0	1	1	2	0	1	2	3	4	5

The deterministic finite automaton $\mathcal{D}(w)$ recognizing the language $\Sigma^* \cdot w$ is defined by $\mathcal{D}(w[1..n]) = (Q, \Sigma, q_0, T, F)$ where $Q = \{0, 1, \ldots, n\}$ is the set of states, Σ is the alphabet, $q_0 = 0$ is the initial state, $T = \{n\}$ is the set of accepting states and $F = \{(i, w[i+1], i+1) \mid 1 \leq i \leq n\} \cup \{(i, a, |Border(w[1..i]a)|) \mid 1 \leq i \leq n \text{ and } a \in \Sigma \setminus \{w[i+1]\}\}$ is the set of transitions. The underlying unlabeled graph is called the *skeleton* of the automaton. We denote by $\delta(i)$ the list $(j \mid (i, a, j) \in F \text{ with } a \in A \text{ and } j \neq 0)$ and by $\delta'(i)$ the list $(j \mid (i, a, j) \in F \text{ with } a \in A \text{ and } j \notin \{0, i+1\})$ for $0 \leq i \leq n$ (see Figure 1). In other words $\delta(i)$ is the list of the targets of the significant transitions leaving state i and $\delta'(i)$ is the list of the targets of the backward significant transitions leaving state i.

The following definition introduces the notion of valid array.

Definition 1 *An integer array $f[1..n]$ is a* valid *array (or is* valid*) if and only if it is the border array of at least one string $w[1..n]$.*

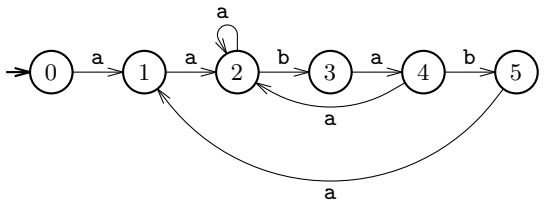


Figure 1: $\mathcal{D}(\texttt{aabab})$: transitions leading to state 0 are omitted. $\delta(4) = (5, 2)$ and $\delta'(4) = (2)$.

The four following definitions show how to represent the notion of border array using trees.

Definition 2 *Given an integer array $f[1..n]$ such that $0 \leq f[i] < i$ we define the relation f on $[-1, n]$ as follows: $i\,f\,j$ if and only if $f[j] = i$ with $0 \leq j \leq n$.*

Definition 3 *$\bar{f}$ is the reflexive, symmetrical and transitive closure of relation f on $[1, n]$.*

Definition 4 *The relation R is defined by $i\,R\,j$ on $[0, n+1]$ if and only if $(i-1)f(j-1)$ with $0 \leq j \leq n+1$.*

Definition 5 *The R-path of j is the sequence of integer $(j_0, j_1, \ldots, j_k)$ such that $j_0\,R\,j$, $j_k = 0$ and $j_{i+1}\,R\,j_i$ for $0 < i < k$.*

Figure 2 illustrates the previous notions on the border array of the string `aabab` used in Fig. 1.

3 Known results

Let $f[1..n]$ be an integer array such that $f[i] < i$ for $1 \leq i \leq n$. For $1 \leq i \leq n$, we define $f^1[i] = f[i]$ and for $f[i] > 0$, $f^\ell[i] = f[f^{\ell-1}[i]]$. We use the following notation: $C(f, i) = (1+f[i-1], 1+f^2[i-1], \ldots, 1+f^m[i-1])$ where $f^m[i-1] = 0$.

In [3], we state the following two necessary and sufficient conditions for an integer array f to be a valid array:

1. $f[1] = 0$ and for $2 \leq i \leq n$, we have $f[i] \in (0) \uplus C(f, i)$;
2. for $i \geq 2$ and for every $j' + 1 \in C(f, i)$ with $j' + 1 > f[i]$, we have $f[j' + 1] \neq f[i]$.

Example 2 Consider the array f from Example 2:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$f[i]$	0	0	1	2	3	0	1	1	2	0	1	2	3	4	5	?

$C(f, 16) = (f[15] + 1, f[f[15]] + 1, f[f[f[15]]] + 1, f^4[15] + 1) = (6, 4, 2, 1)$.

The candidates for $f[16]$ are in $C(f, 16) \uplus (0) = (6, 4, 2, 1, 0)$. Among these values 2 is not valid since $f[4] = 2$.

In [3], we devised an algorithm for verifying if an array f of n integers is valid that checks all the candidates for each $f[i]$ with $1 \leq i \leq n$. This algorithm takes into account the size of the alphabet and when $f[i]$ is equal to 0 it checks if enough letters are available for f to be valid.

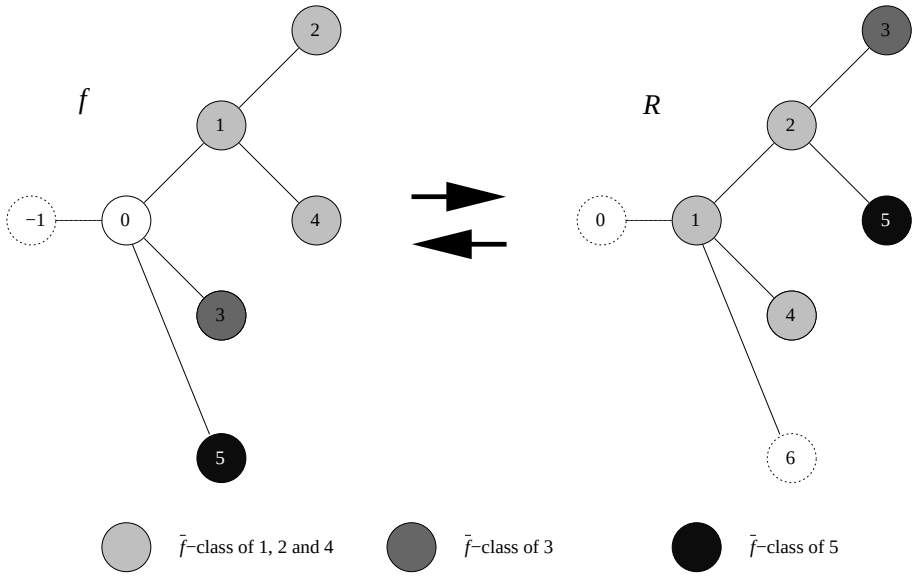


Figure 2: $(i-1)f(j-1)$ iff $i\,R\,j$. $\bar{f}$ is the reflexive, symmetrical and transitive closure of f on $[1,n]$. The R-path of 5 is $(2,1,0)$.

4 New results

In this section we will reformulate the results of [3] and extend the results to the correspondence between the border array f and the skeleton of the deterministic finite automaton recognizing $\Sigma^* \cdot w$ for any string w for which f is the border array.

The next proposition answers the following question: given an integer array $f[1..n]$ with n elements, does there exist a string w such that f is the border array of w?

Proposition 1 *$f[1] = 0$ is the only array with one element that is valid. Let us assume that $f[1..j]$ is valid. Then $f[1..j+1]$ is valid if and only if $f[j+1]$ is the largest element of the $\bar{f}$-class of $j+1$ on the R-path of $j+1$.*

Proof Similar to [3]. □

An example is given Fig. 3 with $f[1..4] = [0,1,0,1]$.

Definition 6 *Two strings with the same length n are f-equivalent if and only if they have the same border array.*

The next proposition answers the following question: given a valid integer array f, what are the f-equivalent strings associated to f?

Proposition 2 *Given a valid integer array f, a string w has f for border array if and only if the following conditions are fulfilled:*

1. *The letters whose indexes are in the same $\bar{f}$-class are identical;*
2. *Two indexes in different $\bar{f}$-classes on a same R-path must correspond to two different letters.*

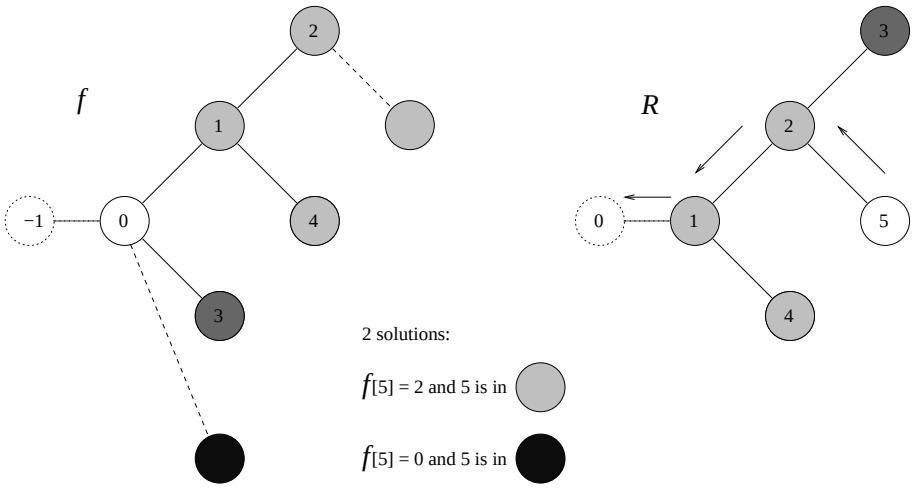


Figure 3: Given $f[1..4]$ a valid array. The R-path of 5 is $(2, 1, 0)$ and 1 is in the same $\bar{f}$-class as 2, so $f[5]$ can only take the values 2 or 0.

Proof Similar to [3]. □

An example is given Fig. 4 with $f[1..5] = [0, 1, 0, 1, 0]$.

The following proposition is rewritten from [7].

Proposition 3 *Let f be an integer array and $1 \leq j \leq n$. If $f[1..n]$ is the border array of a string w and $f[1..j]$ is the border array of a string u then there exists a string v such that $u \cdot v$ is f-equivalent to w.*

The following proposition shows how to build, from a border array f, the skeleton of the automaton recognizing $\Sigma^* \cdot w$ for any f-equivalent string w.

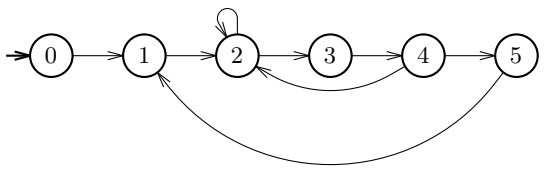
Proposition 4 $\delta(0) = (1)$ *and* $\delta(j) = (j+1) \uplus \delta(f[j]) \cup (f[j+1])$ *for* $1 \leq j < n$ *and* $\delta(n) = \delta(f[n])$.

Proof
Following the definition of the automaton, we have:

$$\begin{aligned}\delta(j) &= (j+1) \uplus (|Border(w[1..j]a)| \mid a \in \Sigma \setminus \{w[j+1]\}) \\ &= (j+1) \uplus (|Border(w[1..j]a)| \mid a \in \Sigma) \cup (|Border(w[1..j+1])|) \\ &= (j+1) \uplus \delta(f[j]) \cup (f[j+1])\end{aligned}$$

□

Example 3 On the following skeleton, that comes from the automaton of Fig. 1:



we indeed have:

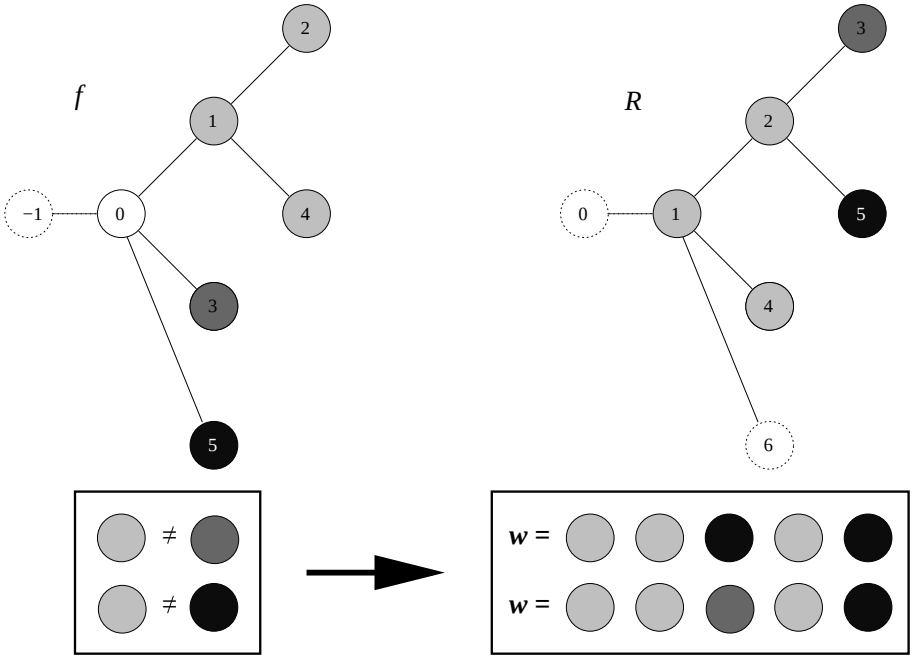


Figure 4: Given $f[1..5]$ a valid array. The letters at positions in $\{1, 2, 4\}$ are equal since they belong to the same $\bar{f}$-class. They must be different from the letters at positions 3 and 5 since they do not belong at the same $\bar{f}$-class and they appear in an R-path. The letter at positions 3 and 5 can be equal or different since they do not appear both in a same R-path.

$j+1$	$\uplus$	$\delta(f[j])$	$\uplus$	$f[j+1]$	$=$	$\delta(j)$
(1)	$\uplus$		$\uplus$		$=$	(1)
(2)	$\uplus$	(1)	$\uplus$	(1)	$=$	(2)
(3)	$\uplus$	(2)	$\uplus$		$=$	(3,2)
(4)	$\uplus$	(1)	$\uplus$	(1)	$=$	(4)
(5)	$\uplus$	(2)	$\uplus$		$=$	(5,2)
	$\uplus$	(1)	$\uplus$		$=$	(1)

The next statement is a corollary of the previous proposition and gives the construction of the border array f from the skeleton of an automaton.

Corollary 1 *For $j > 0$:*

$$f[j+1] = \begin{cases} \delta(f[j]) \uplus \delta(j) & \text{if } \delta(f[j]) \uplus \delta(j) \text{ is not empty,} \\ 0 & \text{otherwise.} \end{cases}$$

Example 4 Using the skeleton of Example 4, we have:

$\delta(f[j])$	$\uplus$	$\delta(j)$	$=$	$f[j+1]$
	$\uplus$	(1)	$=$	0
(1)	$\uplus$	(2)	$=$	1
(2)	$\uplus$	(3,2)	$=$	0
(1)	$\uplus$	(4)	$=$	1
(2)	$\uplus$	(5,2)	$=$	0

It is worth to note that the results of Proposition 4 and Corollary 1 are completely independent from the letters of the underlying string w.

5 New algorithm

The definition of the automaton recognizing $\Sigma^* \cdot w$ gives an efficient algorithm for verifying if an array f of n integers is a valid array. Assuming that $f[1..i]$ is valid, all the values for $f[i+1]$ are in $\delta'(i) \uplus (0)$ and they do not need to be checked. An example is given Fig. 5. Using Proposition 4, the skeleton of the automaton is build on-line during the checking of the array f. If $f[i+1]$ is equal to 0, it is enough to check if the cardinality of $\delta'(i)$ is smaller than the alphabet size s to ensure that f is valid up to position $i+1$. The resulting algorithm is the algorithm DLL(f, n, s) given below. It either outputs a string w of length n on a minimal size alphabet for which f is the border array or the smallest position i for which $f[1..i-1]$ is valid and $f[1..i]$ is not.

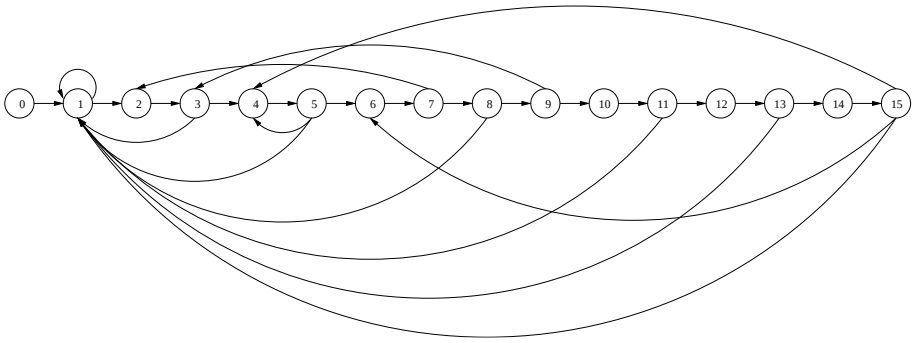


Figure 5: Using the skeleton of the automaton corresponding to the border array of Example 3, it is now easy to see that the candidates for $f[16]$ are in $\delta'(15) \uplus (0) = (6, 4, 1, 0)$.

```
DLL(f, n, s)
 1  if f[1] ≠ 0 then
 2     return f not valid at position 1
 3  δ'(1) ← (1)
 4  w[1] ← σ[1]
 5  for i ← 2 to n do
 6     if f[i] = 0 then
 7        if card(δ'(i − 1)) ≥ s then
 8           return alphabet too small at position i
 9        δ'(i) ← (1)
10        w[i] ← σ[card(δ'(i − 1)) + 1]
11     else if f[i] ∉ δ'(i − 1) then
12           return f not valid at position i
13        δ'(i − 1) ← δ'(i − 1) ⊎ (f[i])
14        δ'(i) ← δ'(f[i]) ⊎ (f[i] + 1)
15        w[i] ← w[f[i]]
16  return w
```

Theorem 1 *The algorithm* DLL(f, n, s) *checks if the array* f *of* n *integers is valid on an alphabet of size* s *in time and space* $O(n)$.

Proof The correctness of the algorithm comes from the definition of the automaton recognizing $\Sigma^* \cdot w$. The time and space linearity comes from the fundamental result that in the automaton, there are only m backward significant transitions [10]. □

We define the delay of the algorithm as the maximal time spent on one element of the array. The next proposition states that the new algorithm lowers the delay from $O(n)$ to $O(\min\{s, n\})$.

Proposition 5 *The delay of the algorithm* DLL(f, n, s) *is* $O(\min\{s, n\})$.

Proof Since $\delta'(i-1)$ contains at most $\min\{s, n\}$ elements, the instructions from line 11 to line 15 of the algorithm DLL can be performed in time

Table 1: Number of distinct border arrays on different alphabets.

i	$B(i)$	$B(i,2)$	$B(i,3)$	$B(i,4)$
1	1	1	1	1
2	2	2	2	2
3	4	4	4	4
4	9	**8**	9	9
5	20	16	20	20
6	47	32	47	47
7	110	64	110	110
8	263	128	**262**	263
9	630	256	626	630
10	1525	512	1509	1525
11	3701	1024	3649	3701
12	9039	2048	8872	9039
13	22,140	4096	21,640	22,140
14	54,460	8192	52,993	54,460
15	134,339	16,384	130,159	134,339
16	332,439	32,768	320,696	**332,438**

$O(\min\{s, n\})$. All the other instructions of the **for** loop of the algorithm DLL can be performed in constant time. □

An algorithm for generating all valid arrays becomes then obvious: all the valid candidates for $f[i]$ are in $\delta'(i-1) \uplus (0)$. We thus have the following result.

Theorem 2 *All the valid arrays of length n on an unbounded alphabet or on an alphabet of size s can be generated in a time proportional to their number.*

6 Counting distinct border arrays

Let $B(n)$ be the number of distinct border arrays of length n on an unbounded alphabet and let $B(n, s)$ be the number of distinct border arrays of length n on an alphabet of size s. Table 1 gives the number of distinct border arrays of length 1 to 16 for an unbounded alphabet and alphabets of size 2 to 4.

Proposition 6 ([7]) $B(n, 2) = 2^{n-1}$.

The result of the previous proposition means that, since there are 2^n different string of length n on a binary alphabet, the f-equivalence on strings on a binary alphabet amounts to an homomorphism on the letters.

Proposition 7 ([7]) $B(j, s) = B(j)$ *for* $j < 2^s$ *and* $s \geq 2$.

Proposition 8 $B(2^s, s) = B(2^s) - 1$ *for* $s \geq 2$. *The missing border array has the following form:* $0..2^0 - 1 \cdot 0..2^1 - 1 \cdots 0..2^{s-1} - 1$. *This border array corresponds to the string* $w_s \cdot \sigma[s+1]$ *(of length* 2^s*) where* w_s *is recursively defined by:* $w_1 = a$ *and* $w_i = w_{i-1} \cdot \sigma[i] \cdot w_{i-1}$ *for* $i > 1$.

Proof We prove by recurrence that the string w_i has borders followed by every letters from $\sigma[1]$ to $\sigma[i]$. This is true for w_1. Let us assume that this is true for w_k with $2 \leq k \leq i-1$. Then $w_i = w_{i-1} \cdot \sigma[i] \cdot w_{i-1}$ has borders w_{i-1} and $w_{i-1} \cdot \sigma[i]$ is a prefix of w_i. □

The string w_i has already been shown to have the largest number of non-deducible periods [2]. It appears in a large number of applications [9].

Example 5 The following array $f[1..16]$ is valid on an alphabet of size at least 5:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$w_4[i]$	a	b	a	c	a	b	a	d	a	b	a	c	a	b	a	e
$f[i]$	0	0	1	0	1	2	3	0	1	2	3	4	5	6	7	0

7 Conclusions and perspectives

In this article we reformulated the notion used in [3] for verifying if a given integer array is a valid array. We extended these results to the relation between the border array f and the skeleton of the deterministic finite automaton recognizing $\Sigma^* \cdot w$. This enables us to design a very efficient algorithm for verifying if a given integer array is a valid array. This algorithm gives an efficient generation method for generating all the distinct border arrays. Moreover we give here some results on their numbers.

In [7, 5] the authors give an upper bound of $B(n)$ the number of distinct border arrays of length n, it would be very interesting to get an exact analytical bound.

Let us recall the function g: $g[j] = \max\{i \mid w[1..i-1]$ suffix of $w[1..j-1]$ and $w[i] \neq w[j]\}$.

We know that $g[j] = \max\{\delta(j-1) - (j)\} = \max\{\delta(f[j-1]) - (f[j])\}$.

Function g is known as the "failure function" of the Knuth-Morris-Pratt string matching algorithm [6]. We intend to study the problem of verifying if a given integer array is a valid "failure function" for the Knuth-Morris-Pratt algorithm. However there does not exist the equivalence of Lemma 3 for g.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

[2] J.-P. Duval. *Contribution à l'étude du monoïde libre*. PhD thesis, Université de Rouen, France, 1980.

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

[4] F. Franěk, S. Gao, W. Lu, P. J. Ryan, W. F. Smyth, Y. Sun, and L. Yang. Verifying a border array in linear time. *Journal on Combinatorial Mathematics and Combinatorial Computing*, 42:223–236, 2002.

[5] S. Gao. New properties of borders and covers of strings. Master's thesis, McMaster University, Canda, 2001.

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

[7] D. Moore, W. F. Smyth, and D. Miller. Counting distinct strings. *Algorithmica*, 23(1):1–13, 1999.

[8] J. H. Morris and V. R. Pratt Jr. A linear pattern-matching algorithm. Technical Report 40, University of California, Berkeley, 1970.

[9] M. Naylor. Abacaba-dabacaba. `http://www.ac.wwu.edu/~mnaylor/abacaba/abacaba.html`.

[10] I. Simon. String matching algorithms and automata. In R. Baeza-Yates and N. Ziviani, editors, *Proccedings of the First South American Workshop on String Processing*, pages 151–157, Belo Horizonte, Brazil, 1993.

[11] W. F. Smyth. *Computing Pattern in Strings*. Addison Wesley Pearson, 2003.