EFFICIENT VALIDATION AND CONSTRUCTION OF BORDER ARRAYS AND VALIDATION OF STRING MATCHING AUTOMATA *

JEAN-PIERRE DUVAL[1], THIERRY LECROQ[1]
AND ARNAUD LEFEBVRE[1]

Abstract. 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. First of all, we show a bijection between the border array of a string w and the skeleton of the DFA recognizing $\Sigma^* w$, called a string matching automaton (SMA). Different strings can have the same border array but the originality of the presented method is that the correspondence between a border array and a skeleton of SMA is independent from the underlying strings. This enables to design algorithms for validating and generating border arrays that outperform existing ones. The validating algorithm lowers the delay (maximal number of comparisons on one element of the array) from $O(|w|)$ to $1 + \min\{|\Sigma|, 1 + \log_2 |w|\}$ compared to existing algorithms. We then give results on the numbers of distinct border arrays depending on the alphabet size. We also present an algorithm that checks if a given directed unlabeled graph G is the skeleton of a SMA on an alphabet of size s in linear time. Along the process the algorithm can build one string w for which G is the SMA skeleton.

Mathematics Subject Classification. 68R15, 68W05.

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 length of the longest border of each prefix of a string w is strongly related to the exact string matching problem: given a string w, find its first occurrence or, more generally, all its occurrences in a longer 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 the border array of at least 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 are 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 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 (see `http://al.jalix.org/Baba/Applet/baba.php`). In the present article we first give a more simple 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 $\Sigma^* w$, called a string matching automaton (SMA). Actually these results are completely independent of w. We then present a new linear time and space on-line algorithm that checks if a given integer array is the border array of at least one string. This algorithm lowers the delay (maximal number of comparisons on one element of the array) from $O(|w|)$ to $1 + \min\{|\Sigma|, 1 + \log_2 |w|\}$ compared to algorithms in [3,4]. An easy extension of this algorithm enables to generate all the distinct border arrays of a given length in linear space and in time proportional to their number.

This study can be useful for generating minimal test sets for various string algorithms. For instance this can be used to test the practical performances, in terms of running times or number of comparisons, of string matching algorithms with strings that have different behaviors rather than with randomly chosen strings that may have the same behavior.

Then using this efficient construction algorithm, we count the number of distinct border arrays for alphabet sizes 2, 3, 4 and we prove results on any alphabet. These last results extend those of [7].

Then we show how to decide whether a given directed unlabeled graph G is the skeleton of a SMA, on an alphabet of size s in linear time, or not. Along the process the algorithm can build a string w for which G is the SMA skeleton.

These methods constitute a first step towards a better understanding of the combinatorics of border arrays and SMA and thus of the combinatorics on words.

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 for validating border arrays. In Section 5 we present the bijection between border arrays and

SMA skeletons. In Section 6 we give our new algorithm for validating border arrays together with its correctness proof. In Section 7 we present results on the number of distinct border arrays. Section 8 presents the linear time method that checks if a given graph G is a SMA skeleton. Finally we give our conclusions and perspectives in Section 9.

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 $w[1 \mathinner{\ldotp\ldotp} n]$, 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 $u \neq w$. A string w can have several borders thus we call *the border* of a string w the longest of its borders. It is denoted by $\mathrm{Border}(w)$. The *border array* f of a string w of length n is defined by: $f[i] = |\mathrm{Border}(w[1 \mathinner{\ldotp\ldotp} i])|$ for $1 \leq i \leq n$. It is also known as the "failure function" of the Morris and Pratt string matching algorithm [8].

Example 2.1. The border array of `ababacaabcababa` is the following:

i	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
$f[i]$	0	0	1	2	3	0	1	1	2	0	1	2	3	4	5

An integer p such that $0 < p \leq |w|$ is a *period* of w if: $w[i] = w[i + p]$ for $i = 0, 1, \ldots, |w| - p - 1$.

The string matching automaton (SMA) $\mathcal{D}(w)$ recognizing the language $\Sigma^* w$ is a DFA defined by $\mathcal{D}(w[1 \mathinner{\ldotp\ldotp} n]) = (Q, \Sigma, q_0, T, F)$ 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 $F = \{(i, w[i+1], i+1) \mid 0 \leq i \leq n-1\} \cup \{(i, a, |Border(w[1 \mathinner{\ldotp\ldotp} i]a)|) \mid 1 \leq i \leq n-1 \text{ and } a \in \Sigma \setminus \{w[i+1]\}\} \cup \{(n, a, |Border(wa)|) \mid a \in \Sigma\}$ is the set of transitions. There exists on elegant on-line construction algorithm for this automaton (see [2]). The underlying unlabeled graph is called the *skeleton* of the automaton. We denote by $\delta(i)$ the list $(j \mid (i, a, j) \in F$ with $a \in \Sigma$ and $j \neq 0)$ and by $\delta'(i)$ the list $(j \mid (i, a, j) \in F$ with $a \in \Sigma$ and $j \notin \{0, i+1\})$ for $0 \leq i \leq n$ (see Figs. 1 and 2). 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. Simon [10] showed that the total number of significant transitions of an SMA of a string of length n is at most $2n$: exactly n forward transitions and at most n backward transitions.

The following definitions introduce the notion of b-valid array and of valid skeleton.

Definition 2.1. An integer array $f[1 \mathinner{\ldotp\ldotp} n]$ is a *b-valid* array (or is *b-valid*) if and only if it is the border array of at least one string $w[1 \mathinner{\ldotp\ldotp} n]$.
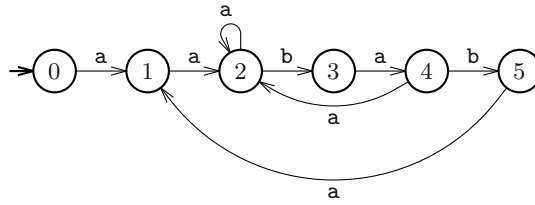
FIGURE 1. $\mathcal{D}(\texttt{aabab})$: transitions leading to state 0 are omitted.
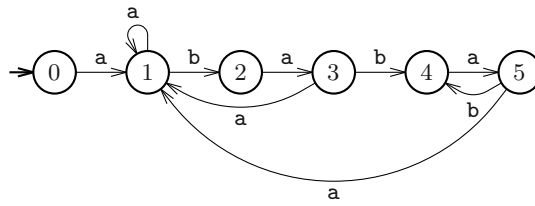$\delta(4) = (5, 2)$ and $\delta'(4) = (2)$.

FIGURE 2. $\mathcal{D}(\texttt{ababa})$: transitions leading to state 0 are omitted.
$\delta(3) = (4, 1)$ and $\delta'(3) = (1)$.

Definition 2.2. Let $f[1 \mathinner{.\,.} n]$ be an integer array such that $f[i] < i$ for $1 \leq i \leq n$.
For $1 \leq i \leq n$, we define $f^0[i] = i$ and for $f[i] > 0$, $f^\ell[i] = f[f^{\ell-1}[i]]$ with $\ell \geq 1$.

Definition 2.3. A directed unlabeled graph is valid if it is the skeleton of a SMA.

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

Definition 2.4. Given an integer array $f[1 \mathinner{.\,.} n]$ such that $0 \leq f[i] < i$ we define
the relation $\mathcal{F}$ on $[-1, n]$ as follows: $0 \mathcal{F} -1$ and $i \mathcal{F} j$ if $f[i] = j$ with $0 \leq j < i \leq n$.

Relation $\mathcal{F}$ is known as the border tree [11].

Definition 2.5. $\bar{\mathcal{F}}$ is the reflexive, symmetrical and transitive closure of relation
$\mathcal{F}$ on $[1, n]$ that is to say $i \bar{\mathcal{F}} j$ if there exist a positive integer $k \neq 0$ and two
positive integers $s, t \geq 0$ such that $f^s[i] = f^t[j] = k$.

$\bar{\mathcal{F}}$ can be seen as a partition of the nodes of the tree induced by the relation
$\mathcal{F}$. Two nodes are in the same $\bar{\mathcal{F}}$-class if their least common ancestor is different
from the root.

Lemma 2.1. *Let f be a b-valid array and w be a string for which f is the border
array. If two integers i and j $(1 \leq i, j \leq |w|)$ are in the same $\bar{\mathcal{F}}$-class and let k be
the smallest element of the $\bar{\mathcal{F}}$-class of i and j then $w[i] = w[j] = w[k]$.*
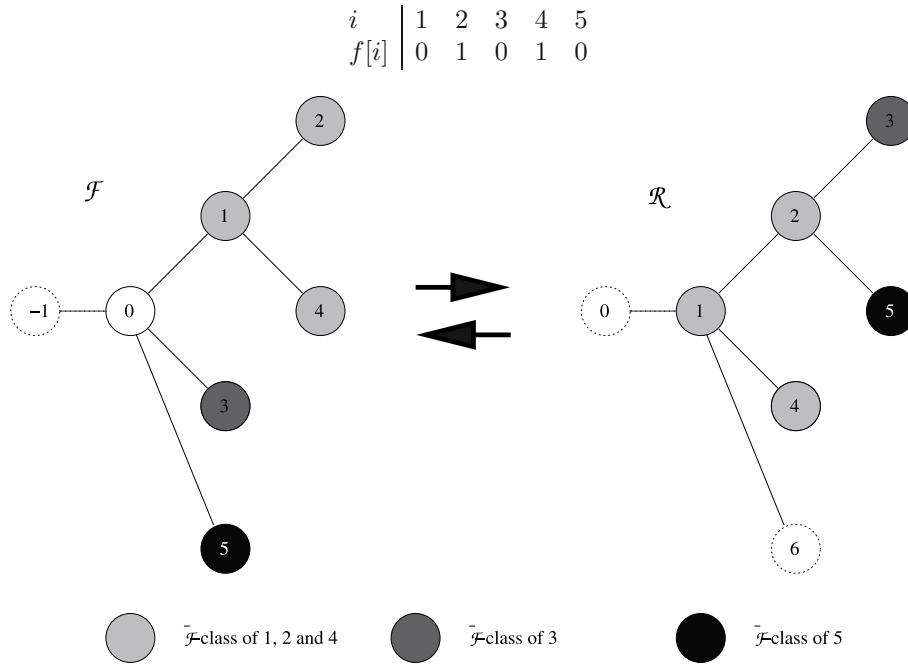
$$
\begin{array}{c|ccccc}
i & 1 & 2 & 3 & 4 & 5 \\
\hline
f[i] & 0 & 1 & 0 & 1 & 0
\end{array}
$$

FIGURE 3. An integer array f and the trees associated with the relations $\mathcal{F}$ and $\mathcal{R}$. The $\mathcal{R}$-path of 5 is $(5, 2, 1, 0)$.

Proof. Since i and k are in the same $\bar{\mathcal{F}}$-class there exist s, t such that $f^s[i] = f^t[k]$. Since k is the smallest element of the $\bar{\mathcal{F}}$-class it means that $t = 0$ and $f^s[i] = k$. Thus $w[1 \mathinner{\ldotp\ldotp} k]$ is a border of $w[1 \mathinner{\ldotp\ldotp} i]$ and in particular $w[i] = w[k]$. The same holds for j. □

Definition 2.6. The relation $\mathcal{R}$ is defined on $[0, n+1]$ by $i \mathcal{R} j$ if and only if $(i-1) \mathcal{F} (j-1)$ with $0 \leq j < i \leq n + 1$.

Definition 2.7. The $\mathcal{R}$-path of j is the strictly decreasing sequence of integers $(j_0, j_1, \ldots, j_k)$ such that $j_0 = j$, $j_i \mathcal{R} j_{i+1}$ for $0 \leq i \leq k - 1$ and $j_k = 0$.

In words, if f is b-valid and if $(j_0, j_1, \ldots, j_k)$ is the $\mathcal{R}$-path of j it means that $w[1 \mathinner{\ldotp\ldotp} j_1 - 1]$ is the border of $w[1 \mathinner{\ldotp\ldotp} j_0 - 1]$, $w[1 \mathinner{\ldotp\ldotp} j_2 - 1]$ is the border of $w[1 \mathinner{\ldotp\ldotp} j_1 - 1]$, ..., $w[1 \mathinner{\ldotp\ldotp} j_{k-1} - 1]$ is the border of $w[1 \mathinner{\ldotp\ldotp} j_{k-2} - 1]$. In other words, $w[1 \mathinner{\ldotp\ldotp} j_1 - 1]$, $w[1 \mathinner{\ldotp\ldotp} j_2 - 1], \ldots, w[1 \mathinner{\ldotp\ldotp} j_{k-1} - 1]$ are all the borders of $w[1 \mathinner{\ldotp\ldotp} j - 1]$.

Figure 3 illustrates the previous notions on the border array of the string `aabab` used in Figure 1.

3. KNOWN RESULTS

Let $f[1 \mathinner{.\,.} n]$ be an integer array such that $f[i] < i$ for $1 \le i \le n$. 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 b-valid array:

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

Example 3.1. Consider the array f from Example 2.1:

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 b-valid that checks all the candidates for each $f[i]$ with $1 \le i \le n$. 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 b-valid.

4. VALIDATION OF BORDER ARRAYS

In this section we will reformulate the results of [3].

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

Proposition 4.1. $f[1] = 0$ *is the only array with one element that is b-valid. Let us assume that $f[1 \mathinner{.\,.} j]$ is b-valid. Then $f[1 \mathinner{.\,.} j+1]$ is b-valid if and only if $f[j+1]$ is a largest element of its $\bar{\mathcal{F}}$-class on the $\mathcal{R}$-path of $j+1$.*

Proof. It can be easily checked that the border array with one element corresponding to a string of length 1 can only contain the value 0. Let us now assume that $f[1 \mathinner{.\,.} j]$ is b-valid. The $\mathcal{R}$-path of $j+1$ is the sequence of integers $(j_0, j_1, \ldots, j_k)$ such that $j_0 = j+1$, $j_i \mathcal{R} j_{i+1}$ for $0 \le i \le k-1$ and $j_k = 0$. Thus $(j_i - 1) \mathcal{F} (j_{i+1} - 1)$ for $0 \le i \le k-1$. Thus $f[j_i - 1] = j_{i+1} - 1$ for $0 \le i \le k-1$. Thus the $\mathcal{R}$-path of $j+1$ is the sequence of integers $(j+1, f[j]+1, f^2[j]+1, \ldots, f^m[j]+1, 0)$ where $f^m[j] = 0$.

If $f[j+1] = f^\ell[j] + 1$ is not the largest element of its $\bar{\mathcal{F}}$-class on the $\mathcal{R}$-path of $j+1$, it means that there exists a k such that $f^k[j]+1$ is on the $\mathcal{R}$-path of $j+1$, and $f[f^k[j]+1] = f[j+1]$ which contradicts the maximality of $f[j+1]$. $\square$
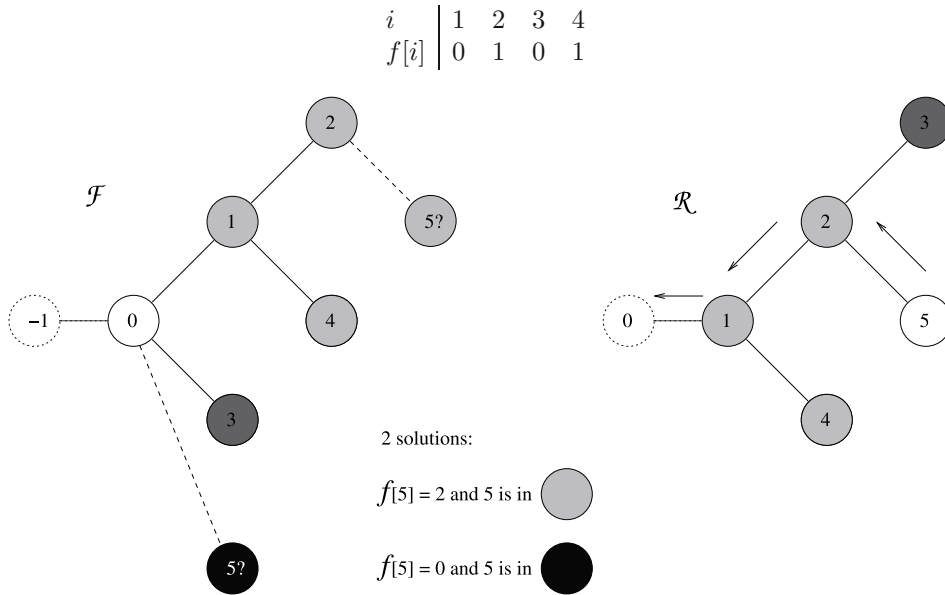
An example is given Figure 4 with $f[1 \mathinner{.\,.} 4] = [0,1,0,1]$.

$$\begin{array}{c|cccc} i & 1 & 2 & 3 & 4 \\ \hline f[i] & 0 & 1 & 0 & 1 \end{array}$$

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

Definition 4.1. Two strings with the same length n are b-equivalent if and only if they have the same border array.

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

Proposition 4.2. *Given a b-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 indices are in the same $\bar{\mathcal{F}}$-class are identical;*
    (2) *two indices in different $\bar{\mathcal{F}}$-classes on a same $\mathcal{R}$-path must correspond to two different letters.*

Proof.
    1. Let i and j be two indices in the same $\bar{\mathcal{F}}$-class. Then there exist three strictly positive integers k, s, t such that $f^s[i] = f^t[j] = k$. Thus, since f is b-valid, it corresponds to the border array of a string w and $w[1 \mathinner{.\,.} k]$ is a border of both $w[1 \mathinner{.\,.} i]$ and $w[1 \mathinner{.\,.} j]$ and thus $w[i] = w[j] = w[k]$.
    2. Let i and j be two indices in different $\bar{\mathcal{F}}$-classes on the $\mathcal{R}$-path of a position k. Assume w.l.o.g. that $i > j$. Then $i \, \mathcal{R} \cdots \mathcal{R} \, j$, thus $(i-1) \, \mathcal{F} \cdots \mathcal{F} \, (j-1)$. Thus $w[1 \mathinner{.\,.} j-1]$ is a border of $w[1 \mathinner{.\,.} i-1]$, but if i and j are not in the

$$\begin{array}{c|ccccc}
i & 1 & 2 & 3 & 4 & 5 \\
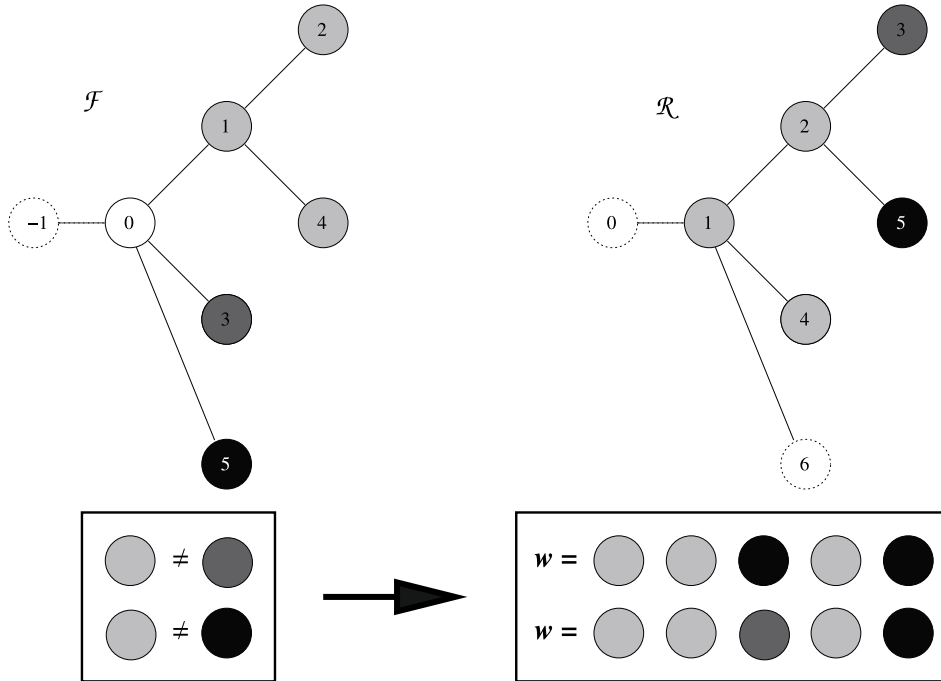\hline
f[i] & 0 & 1 & 0 & 1 & 0
\end{array}$$

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

same $\bar{\mathcal{F}}$-class that means that $w[1 . . j]$ is not a border of $w[1 . . i]$, which implies that $w[i] \neq w[j]$.

This ends the proof of the proposition.   □

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

The following proposition is rewritten from [7].

Proposition 4.3. *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 uv is b-equivalent to w.*   □

5. Bijection between border arrays and SMA skeletons

In this section we explicit the correspondence between the border array f and the skeleton of the deterministic finite automaton recognizing Σ^*w for any string w for which f is the border array.

The following proposition shows how to build, from a border array f, the skeleton δ of the automaton recognizing Σ^*w for any b-equivalent string w.

Proposition 5.1. *Assume that f is a b-valid array then:*

1. $\delta(0) = (1)$;
2. $\delta(j) = (j+1) \ \uplus \ \delta(f[j]) \uplus (f[j+1])$ *for $1 \le j < n$;*
3. $\delta(n) = \delta(f[n])$.

Proof. The correctness of cases 5.1 and 5.3 comes directly from the definition of the SMA (see Algorithm 9.3 in [1]). Following the definition of the automaton, we have:

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

for $1 \le j < n$ which shows case 5.2 and ends the proof of the proposition. $\square$

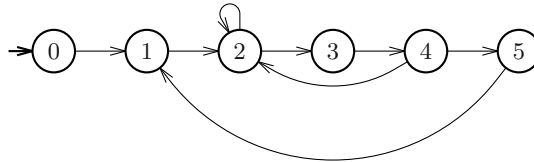**Example 5.1.** On the following array:

i	1	2	3	4	5
$f[i]$	0	1	0	1	0

we indeed have:

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

This gives the following skeleton, that comes from the automaton of Figure 1:

The next proposition gives the construction of the border array f from the skeleton δ of a SMA.

Proposition 5.2. *For $j > 0$:*

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

Proof. Recall item 2 in Proposition 5.1:

$$\delta(j) = (j+1) \ \uplus \ \delta(f[j]) \uplus (f[j+1]) \text{ for } 1 \le j < n. \tag{2}$$

Note that $(j+1) \notin \delta(f[j])$. We distinguish three cases:

- $f[j+1] = f[j]+1$. Since $f[j]+1 \in \delta(f[j])$, from (2), we have that $f[j+1]$ is the unique element of $\delta(f[j]) \uplus \delta(j)$.
- $f[j+1] \ne f[j]+1$ and $f[j+1] \ne 0$. Since $f[j+1] \in \delta(f[j])$, from (2), we have that $f[j+1]$ is the unique element of $\delta(f[j]) \uplus \delta(j)$.
- $f[j+1] \ne f[j]+1$ and $f[j+1] = 0$. $f[j+1] \notin \delta(f[j])$ and, from (2), $\delta(f[j]) \uplus \delta(j)$ is empty. $\square$

Example 5.2. Using the skeleton of Example 3, we have:

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

It is worth to note that the results of Propositions 5.1 and 5.2 are completely independent of the letters of the underlying string w, thus:

Theorem 5.1. *Propositions 5.1 and 5.2 define a bijection between border arrays and SMA skeletons.* $\square$

6. Checking the validity of border arrays

The definition of the SMA gives an efficient algorithm for verifying if an array f of n integers is a b-valid array. Assuming that $f[1 \mathbin{..} i]$ is b-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 Figure 6. Using Proposition 5.1, the skeleton of the automaton is build on-line during the checking of the array f. If f is b-valid it is possible to compute a string w for which f is the border array. If $f[i]$ is equal to 0, it is enough to check if the cardinality of $\delta'(i-1)$ is smaller than the alphabet size s to ensure that f is b-valid up to position i.

The result is Algorithm CheckArray(f, n, s) below. It either outputs *true* if the array f is b-valid or the smallest position i for which $f[1 \mathbin{..} i-1]$ is b-valid and $f[1 \mathbin{..} i]$ is not. Along the line it builds a string w of length n on a minimal size alphabet for which f is the border array.
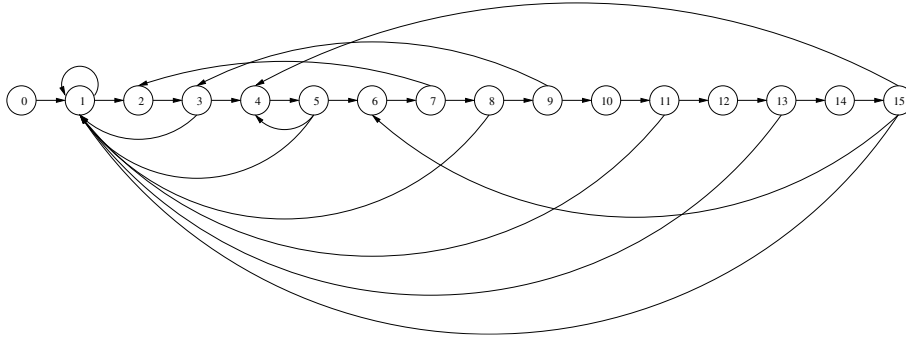
FIGURE 6. Using the skeleton of the automaton corresponding to the border array of Example 2, it is now easy to see that the candidates for $f[16]$ are in $\delta'(15) \uplus (0) = (6, 4, 1, 0)$.

$\text{CHECKARRAY}(f, n, s)$

1	**if** $f[1] \neq 0$ **then**	$\triangleright$ validity
2	**return** f not b-valid at position 1	$\triangleright$ validity
3	$\delta'(1) \leftarrow (1)$	
4	$w[1] \leftarrow \sigma[1]$	$\triangleright$ string
5	**for** $i \leftarrow 2$ **to** n **do**	
6	**if** $f[i] = 0$ **then**	
7	**if** $card(\delta'(i-1)) \geq s$ **then**	$\triangleright$ alphabet
8	**return** alphabet too small at position i	$\triangleright$ alphabet
9	$\delta'(i) \leftarrow (1)$	
10	$w[i] \leftarrow \sigma[card(\delta'(i-1)) + 1]$	$\triangleright$ string
11	**else**	
12	**if** $f[i] \notin \delta'(i-1)$ **then**	$\triangleright$ validity
13	**return** f not b-valid at position i	$\triangleright$ validity
14	$\delta'(i-1) \leftarrow \delta'(i-1) \uplus (f[i])$	
15	$\delta'(i) \leftarrow \delta'(f[i]) \uplus (f[i]+1)$	
16	$w[i] \leftarrow w[f[i]]$	$\triangleright$ string
17	**return** $true$	

Theorem 6.1. *When applied to an integer array $f[1 \mathinner{.\,.} n]$ and an alphabet of size s:*

- *The algorithm* CHECKARRAY *runs in time and space $\Theta(n)$.*
- *If the array f given as input of the algorithm* CHECKARRAY *is a b-valid array at index $i-1$ but not at index i, the algorithm stops and returns "f not b-valid at position i". The lines {**alphabet**} and {**string**} can be deleted without changing this result.*

- *If there exists a string for which $f[1 .. i-1]$ is the border array and there is none at index i with an alphabet of size s, the algorithm* CHECKARRAY *stops and returns "alphabet too small at position i". Lines {**string**} can be deleted without changing this result. If the array f is b-valid, lines {**validity**} can also be deleted.*
- *As long as $f[1 .. i]$ is valid, the algorithm* CHECKARRAY *builds a string $w[1 .. i]$ on a minimal size alphabet for the border array $f[1 .. i]$. Lines {**validity**} can be deleted without changing the construction of the string. It is clear that if f is not b-valid, it is not the border array of the string which is built by the algorithm.*

Proof. The correctness of the computation of line 9 comes from Case 5.1 of Proposition 5.1. The correctness of the computation of line 14 comes from Case 5.2 of Proposition 5.1. The correctness of the computation of line 15 comes from Case 5.3 of Proposition 5.1.

The time and space linearity comes from the fundamental result that in the SMA, there are only m backward significant transitions [10].   □

It can be noticed that when computing the border array of string w of length n the number of comparisons between letters of w is $2n - 3$ in the worst case. This bound is reached for $w = a^{n-1}b$. When executing the algorithm CHECKARRAY(f, n, s) the number of comparisons of elements of f are performed lines 1, 6 and 12. There can be only one comparison in Line 1, $n - 1$ comparisons in Line 6, and n comparisons overall in Line 12. Together this gives a total upper bound of $2n$ comparisons. However the worst case on the number of backward significant transitions in the SMA is reached for strings of the form ab^{n-1} where there is a backward transition leaving state 1. Thus the maximal number of comparisons on elements of f performed by the algorithm CHECKARRAY(f, n, s) is $2n - 1$. This bound is reached for $f = [0, 1, 2, \ldots, n - 2, n - 1]$.

We now define the delay of the algorithm as the maximal number of comparisons on $f[i]$ for each i with $1 \le i \le n$. The next proposition states that the new algorithm lowers the delay from $O(n)$ (see [3,4]) to $1 + \min\{s, 1 + \log_2 n\}$.

Proposition 6.1. *The delay of the algorithm* CHECKARRAY(f, n, s) *is* $1 + \min\{s, 1 + \log_2 n\}$.

Proof. $f[1]$ is compared once line 1. For each i, $2 \le i \le n$, $f[i]$ is compared lines 6 and 12. There is exactly one comparison on line 6. When $f[i]$ is processed line 12, the skeleton of the automaton is built up to state $i - 1$. Thus $\delta'(i - 1) = \delta(i - 1)$. Since $\delta(i - 1)$ contains at most $\min\{s, 1 + \log_2 n\}$ elements (see Prop. 2.7 in [5]), $\delta'(i - 1)$ contains at most $\min\{s, 1 + \log_2 n\}$ elements. Consequently, the maximal number of comparisons is $1 + \min\{s, 1 + \log_2 n\}$.   □

An algorithm for generating all b-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 6.2. *All the b-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 and in linear space.*

TABLE 1. Number of distinct border arrays on different alphabet sizes.

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**
17	824 735	65 536	792 265	824 731
18	2 051 307	131 072	1 962 407	2 051 291
19	5 113 298	262 144	4 872 223	5 113 246
20	12 773 067	524 288	12 123 877	12 772 899
21	31 968 041	1 048 576	30 230 923	31 967 537
22	80 152 901	2 097 152	75 528 071	80 151 415
23	201 297 338	4 194 304	189 039 446	201 293 090
24	506 324 357	8 388 608	473 956 301	506 312 374
25	1 275 385 911	16 777 216	1 190 195 672	1 275 352 669
26	3 216 901 194	33 554 432	2 993 316 684	3 216 809 897
27	8 124 150 323	67 108 864	7 538 797 541	8 123 902 127

7. 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 7.1 [7]. $B(n, 2) = 2^{n-1}$.

Proof. By recurrence on n. $B(1, 2) = 1$. Let $f[1 .. n]$ be b-valid with δ the corresponding skeleton. For an alphabet of size 2, $\delta(n)$ contains at most 2 elements. Consider the three possible cases:

- $\delta(n) = \{i_1, i_2\}$: $f[n + 1]$ can only be equal either to i_1 or to i_2.

- $\delta(n) = \{i\}$: $f[n+1]$ can only be equal either to i or to 0.
- $\delta(n) = \emptyset$: impossible since $\delta(n) = \delta(f[n])$ (see Prop. 5.1). $\square$

Indeed there are 2^n different strings of length n on a binary alphabet $\{a, b\}$, and 2^{n-1} distinct border arrays of length n since the b-equivalence on strings on a binary alphabet amounts to an homomorphism h such that $h(a) = b$ and $h(b) = a$. This is not the case on larger alphabets, for instance *abb*, *abc*, *cab* have the same border array but there is no letter homomorphism between these strings.

Proposition 7.2 [7]. $B(j, s) = B(j)$ for $j < 2^s$ and $s \geq 2$.

Proof. By recurrence on s. $B(1, 2) = B(1) = 1$, $B(2, 2) = B(2) = 2$ and $B(3, 2) = B(3) = 4$. Assume that $B(j, k) = B(j)$ for $j < 2^k$ for $k \leq s$. By recurrence assumption the first occurrence of $\sigma[k+1]$ in strings corresponding to border arrays counted by $B(j, k+1)$ is larger than 2^k. Suppose now that $B(j, s+1) < B(j)$. This means that the letter $\sigma[s+2]$ is required to build all the distinct border arrays. Let w be the string that corresponds to a border array that requires $s+2$ letters. The letter $\sigma[s+2]$ can only occur at a position i greater or equal to 2^s. And this can only happen if the strings $w[1 .. i-1]\sigma[1]$, $w[1 .. i-1]\sigma[2]$, ..., $w[1 .. i-1]\sigma[s+1]$ have all non-empty borders. Let ℓ be length of the border of $w[1 .. i-1]\sigma[s+1]$ in w. Then $\ell \leq 2^s$. But this implies that w has a period $i - \ell < 2^s$ and that $w[\sigma[s+1]]$ occurs in the first $2^s - 1$, which is a contradiction. $\square$

Proposition 7.3. $B(2^s, s) = B(2^s) - 1$ for $s \geq 2$. The missing border array has the following form: $[0, \ldots, 2^0 - 1, 0, \ldots, 2^1 - 1, \ldots, 0, \ldots, 2^{s-1} - 1]$. This border array corresponds to the string $w_s \sigma[s+1]$ (of length 2^s) where w_s is recursively defined by: $w_1 = \sigma[1]$ and $w_i = w_{i-1}\sigma[i]w_{i-1}$ 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 $w_i = w_{i-1}\sigma[i]w_{i-1}$ has all the borders of w_{i-1} and $w_{i-1}\sigma[i]$ is a prefix of w_i. $\square$

The string w_i has already been shown to have the largest number of non-deducible periods [5]. It appears in a large number of applications [9].

Example 7.1. The following array $f[1 .. 16]$ is b-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

8. VALIDATION OF A STRING MATCHING AUTOMATON

Corollary 5.2 gives a method to check if a given directed graph $G = (V, E)$, can be the skeleton of a SMA. The graph of $n + 1$ vertices numbered from 0 to n is supposed to satisfy the following conditions:

- no edge ending in vertex 0;
- $(0, 1, \ldots, n)$ is a unique simple path from 0 to n.

Since a skeleton of SMA with $n+1$ states has at most $2n$ transitions, a given graph can be rejected or numbered, according to the two previous conditions, in linear time $O(n)$.

Then it is possible to use Corollary 5.2 to check if G can be the skeleton of a SMA. For each state j, the difference $D = \delta(f[j]) - \delta(j)$ is computed: when it is empty or equal to a singleton then G is a skeleton up to state j. If D has more than one element then G is not a skeleton. If D is empty then $f[j + 1]$ is set to 0. If D is a singleton $\{i\}$ then $f[j + 1]$ is set to i.

Since only one value of f is needed at a time it is even possible to perform the checking without building an integer array f during the process.

It is also possible to build a string during the checking process: when $f[j + 1]$, stored in a variable k, is different from 0 then $w[j + 1]$ is set to $w[k]$ and when $f[j + 1]$ is equal to 0 then $w[j + 1]$ is set to $\sigma[card(\delta(j))]$.

It is even possible to build a string on an alphabet of size s: when $f[j + 1]$ is different from 0 then the letter $w[j + 1]$ already occurred before but when $f[j + 1]$ is equal to 0 then $card(\delta(j))$ has to be smaller or equal to s. Algorithm CHECKGRAPH(δ, n, s) below integrates all these results.

CHECKGRAPH(δ, n, s)

```
 1  k ← −1
 2  for j ← 0 to n − 1 do
 3      D ← δ(k) − δ(j)
 4      if D = ∅ then
 5          k ← 0
 6          if card(δ(j)) ≤ s then              ▷ alphabet
 7              w[j + 1] ← σ[card(δ(j))]        ▷ string
 8          else return G not a skeleton at vertex i   ▷ alphabet
 9      else
10          if card(D) = 1 then                 ▷ validity
11              k ← i such that D = {i}
12              w[j + 1] ← k                     ▷ string
13          else return alphabet too small at position j
14  if δ(n) = δ(k) then                          ▷ validity
15      return true                              ▷ validity
16  else  return false                           ▷ validity
```

An example is shown Figure 7.
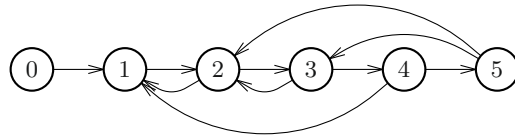
k	-1	0	1	2
$\delta(k)$	$\emptyset$	(1)	(2)	$(1,3)$
j	0	1	2	3
$\delta(j)$	(1)	(2)	$(1,3)$	$(2,4)$
$\delta(k) - \delta(j)$	$\emptyset$	(1)	(2)	$(1,3)$
k	0	1	2	fail
$w[j+1]$	a	a	a	

FIGURE 7. The above graph is a skeleton of a SMA up to vertex 2 but not up to vertex 3 since $\delta(f[3] = 2) - \delta(3) = (1,3)$ possesses two elements. Overall it is not a skeleton of a SMA.

Theorem 8.1. *When applied to a graph G with e edges and v vertices:*

- *The algorithm* CHECKGRAPH *runs in time and space $O(e + v)$.*
- *If the graph G given as input of the algorithm* CHECKGRAPH *is a valid skeleton up to vertex $j - 1$ but not up to vertex j, the algorithm stops and returns "G not a skeleton at vertex i". The lines {**alphabet**} and {**string**} can be deleted without changing this result.*
- *If there exists a string for which the first $j - 1$ vertices of G form a valid skeleton and there is none for the j first vertices with an alphabet of size s, the algorithm* CHECKGRAPH *stops and returns "alphabet too small at position j". Lines {**string**} can be deleted without changing this result. If the graph G is a SMA skeleton, lines {**validity**} can also be deleted.*
- *When the graph G is a valid skeleton, the algorithm* CHECKGRAPH *builds a string $w[1 \mathinner{.\,.} i]$ on a minimal size alphabet for SMA skeleton G. Lines {**validity**} can be deleted without changing the construction of the string.*

Proof. The correctness of the algorithm comes from Corollary 5.2. The time complexity comes from the fact that each vertex and each edge are processed only once. □

Corollary 8.1. *The skeleton of a SMA of n states can be checked in linear time.*

Proof. The result comes from the fact that a SMA has n forward transitions and at most n backward significant transitions [10]. □

9. Conclusions and perspectives

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

Furthermore we presented an algorithm that can check if a given graph G whose vertices are already ordered can be the skeleton of the SMA of at least one string w on an alphabet of size s in linear time in the size of the graph. The method also enables to exhibit, with the same complexity, a string w such that G is the skeleton of the SMA of w.

Let us recall the function g: $g[j] = \max\{i \mid w[1\mathinner{..}i-1]$ suffix of $w[1\mathinner{..}j-1]$ 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 Proposition 4.3 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] M. Crochemore, C. Hancart and T. Lecroq, *Algorithms on Strings*. Cambridge University Press (2007).

[3] J.-P. Duval, T. Lecroq and A. Lefebvre, Border array on bounded alphabet. *J. Autom. Lang. Comb.* **10** (2005) 51–60.

[4] F. Franĕk, S. Gao, W. Lu, P.J. Ryan, W.F. Smyth, Y. Sun and L. Yang, Verifying a border array in linear time. *J. Combin. Math. Combin. Comput.* **42** (2002) 223–236.

[5] C. Hancart, *Analyse exacte et en moyenne d'algorithmes de recherche d'un motif dans un texte*. Ph.D. thesis. Université Paris 7, France (1993).

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

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

[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 *Proceedings of the First South American Workshop on String Processing*, edited by R. Baeza-Yates and N. Ziviani, Belo Horizonte, Brazil (1993) 151–157

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