Use of N-ary trees for pattern extraction from a stream of data

Lorenzo Cioni

Dipartimento di Informatica, Università di Pisa, Pisa 56127, Italy
e-mail: lcioni@di.unipi.it

Abstract. The paper presents the use of an $N-ary$ tree for the extraction of certain patterns from a sequential stream of data. The data of the stream belong to N disjoint categories whereas the patterns are defined with an ex-ante fixed set of K rules. The paper presents an algorithm written using a pseudo-code and shows an application for the extraction of particular patterns from a stream of written Italian.

1 Introduction

Let us suppose we have a stream **F** of data whose elements σ_i belong to N disjoint categories $\mathbf{C}=\{C_i \mid i=0,\ldots,N-1\}$ so that we have an alphabet $\Sigma=\{\sigma_i : i=1,\ldots,l\}$ with a partitioning[1] with $\mid \Sigma \mid = l$.
Our aim is the extraction from **F** of certain patterns as specified by an ex-ante fixed set of K rules $\mathbf{R}=\{R_j \mid j=0,\ldots,K-1\}$. In this paper we present an algorithm that allows such extraction through the use of a generalisation of the concept of binary tree, the $N-$ary tree, and we show an application to written Italian. A companion paper ([1]) is devoted to the use of a similar algorithm for data segmentation.

2 Binary and $N-$ary trees

A tree ([2], [4]) is usually defined as a graph without cycles where[2] we have a set of nodes N and a set of arcs A with $\mid N \mid = n$ and $\mid A \mid = m$, we have a special node called **root**, any pair of nodes is connected by exactly one path and a tree with n nodes has exactly $m=n-1$ arcs. A node can be an **inner** node if it is the root of a non empty sub-tree otherwise it is a leaf. For any node we define its degree as the number of its direct descendants and say that a tree is **balanced** if every inner node has the same number of descendants otherwise is

[1] We have a classical partitioning: $\Sigma = \cup_{i=0}^{N-1} C_i$ and $C_i \cap C_j = \emptyset$ for all $i \neq j$, $i,j=0,\ldots,N-1$. It is obvious that $l > N$.

[2] The symbol N is used to denote the set of the nodes, the number of the categories and the number of the descendants of a node. Context makes clear, in each case, which is the correct meaning. The cardinality of N (and so the number of nodes of the tree) will be denoted with n as the number of input data. Again, the context will make clear which is the correct meaning.

said **unbalanced**. In a **balanced binary tree** every node is either a leaf (and it has a degree 0) or it has a degree equal to 2. A **balanced $N-$ary tree** ([2], [4]), analogously, is a tree whose nodes have degree either 0 (leaves) or N. In the latter case, the descendants are numbered, from left to right, form 0 to $N-1$. In what follows we are going to consider also unbalanced $N-$ary trees.

3 $N-$ary trees and pattern extraction

An $N-$ary tree can be used for **pattern extraction** i. e. for the extraction from **F** of certain pre-defined patterns. Every such pattern corresponds to a string, with repetitions, of category identifiers. If we identify the arcs of the tree with labels $C_i \in \mathbf{C}$ that are distinct for nodes at the same level[3], we can associate to each leaf of the tree a distinct string of category identifiers. Every such a string, that can contain repeated category identifiers, identifies a **meaningful succession** of categories whose occurrence allows the use, on **F**, of one of the extraction rules. According to this definition we can have the pattern $C_0C_0C_iC_{N-1}$ corresponding to a leaf at level 4.
Let us suppose we have, at a certain point of the processing, a given input data stream **F** and that, among the others, we are looking for a pattern of the type $C_iC_{N-1}C_j$. If $a_1a_2a_3a_4 \ldots a_n$ represents the initial part of **F** and if $a_2 \in C_i$, $a_3 \in C_{N-1}$ and $a_4 \in C_j$ we have identified a pattern that is instance[4] of one of the searched for patterns. During the tree traversal we have therefore reached one of the leaves.
Now, since during the scan of both the input string and the tree we have reached a leaf, we have identified a *meaningful succession* of categories to which it corresponds an extraction rule: from the input data stream **F** the instance pattern $a_2a_3a_4$ is therefore extracted and written on the output data stream, according to a format to be specified, whereas on the input stream $a_5 \ldots a_n$ the algorithm is recursively applied. The algorithm ends when there are no more data in **F**.
The algorithm does not really need the tree to be statically built since such a tree can be dynamically built if the input data belong to N disjoint and a priori defined categories. All it is needed is, indeed, only the weight of a node and such a weight can be evaluated as a running sum of the weight of each category to which the data belong (cf. section 5.1).

4 Extraction rules

4.1 Formal definition

An **extraction rule** (simply **rule**) can be expressed as:

$$R_j = if\ \mathbf{condition}_j\ then\ \mathbf{action}_j \tag{1}$$

[3] The level of a node in a tree is the length of the path, as a number of arcs, from that node to the root, that has level 0.

[4] A pattern is a set of categories whereas an instance pattern is a string σ every element of which orderly belongs to each category of the pattern.

(for $j = 0, \ldots, K-1$) where $\mathbf{condition}_j$ represents a logical condition that, in our context, is a type test on substrings of $\mathbf{F}$. If the condition is verified the specified $\mathbf{action}_j$ can be applied to $\mathbf{F}$. The action turns in the extraction from $\mathbf{F}$ of a given instance pattern that is written on the output stream for further processing. In practice, every rule R_j is univocally identified by a numeric identifier $r_j \in [r_{min}, r_{max}]$ so that $\mathbf{condition}_j$ is implemented as an equality test between a current identifier (to be defined shortly) and the various r_j: in the only positive case the action associated to the rule who triggers is executed. Since the aforesaid range contains $\hat{k} = r_{max} - r_{min} + 1$ integer values whereas the rules are $K < \hat{k}$ we have that, in general, not all the values of r_j are associated to a rule but some are associated to a *null* value[5].

4.2 Characterisation and properties

When we define the extraction rules we have to face a trade−off between **quantity** and **complexity** since we can realistically devise either many simple rules or a restricted set of complex rules[6].
In the former case we have fine grained rules and categories and each rule simply defines the position of every instance pattern within $\mathbf{F}$. In the latter case we have coarse grained rules and categories and each rule contains a set of sub cases for the detection of instance patterns. We will briefly describe the second approach in section 6. From here on we, therefore, suppose to be in the first scenario so that the set of rules forms a **suitable set** and can be implemented with an **array** of integers $R[]$ of size $r_{max} - r_{min} + 1$. The indexing scheme is $id = r_j - r_{min}$.
Rules are, moreover, characterised by the following properties: **uniformity** and **completeness**. Uniformity means that every rule works on the same number of input symbols whereas completeness means that to every $r_j \in [r_{min}, r_{max}]$ is associated a rule (cf. further on). In reference to the $N-$ary tree, completeness means that we have the same number of descendants for every inner node whereas uniformity means that all the leaves are at the same level. Usual cases are: completeness and uniformity, non completeness but uniformity. If we have completeness but non uniformity we have conflicting rules where shorter rules (that should correspond to inner nodes), owing to rule according to which we choose among them, hide the longer ones (though they are associated to leaves). We disregard this case. We are going to examine the last case of non completeness and non uniformity in section 6. In our cases, if the rules form a suitable set, we have that the generic element $R[id]$ contains either a *null* value, if to it there corresponds no rule, or an integer value *delta* if to it there corresponds

[5] *null* is a mnemonic for -1 and, therefore, represents an integer value that cannot identify any instance pattern.

[6] The other two cases are: few simple rules and many complex rules. The former case has a little utility and can only be used in very special cases whereas the latter case can be computationally heavy. We are not going to examine such possibilities any further in this paper.

a rule (cf. section 5.1). Every rule can turn, therefore, in the extraction of an instance pattern from **F** and in its insertion in the output data stream.
Before stepping to the next section, where we present the pseudo code of the algorithm and show how the current identifier is evaluated, we give here one simple example.
Example. *Many simple rules, uniformity and completeness.* Let us suppose we have $\Sigma = \{a, b, c\}$ and $C_0 = \{b\}$, $C_1 = \{c\}$, $C_2 = \{a\}$ so that $N = 3$. If we look for patterns of length 2 we have to consider the elements of $\{b, c, a\} \times \{b, c, a\}$ so that we may have the following rules[7] where *curr_string* identifies the data on which rules act:

1. $R_0 = if\ curr\_string ==" bb"\ then\ write(out," i - bb \setminus n");$
2. $R_1 = if\ curr\_string ==" cb"\ then\ in :=" b" + in;$
3. $R_2 = if\ curr\_string ==" ab"\ then\ in :=" b" + in;$
4. $R_3 = if\ curr\_string ==" bc"\ then\ in :=" c" + in;$
5. $R_4 = if\ curr\_string ==" cc"\ then\ write(out," i - cc \setminus n");$
6. $R_5 = if\ curr\_string ==" ac"\ then\ in :=" c" + in;$
7. $R_6 = if\ curr\_string ==" ba"\ then\ in :=" a" + in;$
8. $R_7 = if\ curr\_string ==" ca"\ then\ in :=" a" + in;$
9. $R_7 = if\ curr\_string ==" aa"\ then\ write(out," i - aa \setminus n");$

In this case we define as patterns to be extracted pairs of equal symbols.
If we code C_2 with 2, C_1 with 1 and C_0 with 0 we have the following corresponding identifiers (evaluated from left to right): $r_0 = r_{min} = 0$, $r_1 = 1$, $r_2 = 2$, $r_3 = 3$, $r_4 = 4$, $r_5 = 5$, $r_6 = 6$, $r_7 = r_{max} = 7$. Moreover we have the following vector of delta values: $R[] = [2, 0, 0, 0, 2, 0, 0, 0, 2]$. If we denote with p the current identifier we have that the first of the above rules can be rewritten as:

$$if\ p == r_0\ then\ write(out," i - bb \setminus n"); \tag{2}$$

and the same holds for the others. In practice such rules are coded within an ad hoc procedure under the following form:

$$if\ p == 0\ then\ delta = 2; \tag{3}$$

If as **F** we have *aabacbcababbaccac*, we get (disregarding newlines) $0 - aa$, $10 - bb$ and $13 - cc$.
If we have uniformity but not completeness we can have:

1. $R_0 = if\ curr\_string ==" bb"\ then\ write(out," i - bb \setminus n");$
2. $R_1 = if\ curr\_string ==" aa"\ then\ write(out," i - aa \setminus n");$

[7] With *in* and *out* we define the input and the output stream respectively whereas "+" is, in this case, a classical concatenation operator between a string and a stream, in this case represented by the input string. With i we define the global starting position of the pattern on the input stream and $\setminus n$ is the newline character.

so that we have: $R[] = [2, null, null, null, null, null, null, null, 2]$. If as **F** we have *aabacbcababbaccac*, we get (disregarding newlines) $0 - aa$ and $10 - bb$ (cf. section 5.2).
In all the cases we define two integers $trigIn$ and $trigOut$ as, respectively, the minimum and maximum number of input data to be considered in the evaluation of the rules (in the present cases they are both equal to 2). Such values are statically fixed, given the structure of the rules. If we have, for instance, $trigIn = 3$ and $trigOut = 5$ we have that the rules involve strings of at least three characters but of five characters at the most. In this case, the first two characters are not disregarded and contribute to the identification of one of the rules and, at the same time, of one of the patterns. On the other hand characters beyond the $trigOut-$th cannot contribute but cannot be fully disregarded, as it is shown by rules such as R_1 and the like.

5 The algorithm

5.1 Introduction

The algorithm accepts a stream of data **F** and a set of rules **R** (input) and produces (output) a stream of instance patterns. Each instance pattern is written out on a distinct line preceded by the value of the pointer that defines its global position within **F**.
We remind that **F** is made of n elements, with n not known a priori but finite, and the algorithm scans it sequentially with the aid of a pointer $m \in [0, n-1]$. The algorithm executes two steps of matching:

1. a weighted **category matching** step that assigns every element of **F** to one of the N categories and evaluates a current identifier or weight for the substring scanned up to that point;
2. a **rule matching** step that, according to the weight associated to the current substring, locates the rule to be applied for the processing of **F**.

If $car_m \in \Sigma$ is the current element on **F** (in relative position m) and p denotes the current weight of the current substring, we have:

$$p = p + iN^m \tag{4}$$

where p and m are put to 0 at every recursive call or general reset and $i = match\_category(car_m)$ identifies the category to which the current element belongs. Since we have N categories and each category contains m_i elements $match\_category(car_m)$ looks for car_m among $\sum_{i=0}^{N-1} m_i = l$ elements and returns the index of the category to which car_m belongs. The value of p from (4) represents the **current identifier** and it is used to locate the rule to be applied, if any. If such a rule exists it allows the definition of a **shift** value $delta$ that can assume three values: 0, j or $null$. Formally we have:

$$delta = match\_rule(p) \tag{5}$$

Under the condition $trigIn \leq strlength(curr\_string) \leq trigOut$ we have the following cases.

1. If $delta == j$ we have a pattern of length j from the current position that must be written out preceded by its global position and with a trailing newline. In this case we have a recursive call in the form of tail recursion ([3]).
2. If $delta == 0$ we have no pattern but one character must be put back on the input stream and a general reset follows.
3. If $delta == null$ none of the rules is associated to the current value of p so that the algorithm must go on scanning **F** till one of the two preceding cases occurs or $strlength(curr\_string) > trigOut$.

Until $strlength(curr\_string) < trigIn$ the algorithm goes on with input scanning one character at a time whereas if $strlength(curr\_string) > trigOut$ we have that no rule can be applied. This case can occur only if rules do not satisfy completeness. In this case the algorithm behaves as in the case $delta == 0$.

5.2 The structure

The algorithm has the following structure[8].:

```
R=read_rules(rules_file); \\loads the set of rules in the array
                          \\R[] from a text file rules_file
trigIn=find_min(R);
trigOut=find_max(R);
C=read_categories(categories_file); \\loads the set of categories in the
                            \\array C[] from a text file categories_file
N=size_of(C); \\evaluates the number of the categories
pt=0; \\global position of patterns within the input stream
procedure extract(in, pt)
{
     <<initial_step>>
     while not EOF do
     {
       <<current_step>>
       if(delta > 0)
       {
         <<extract_data>>
         extract(in, pt);
       }
```

[8] We remind that $trigIn$ and $trigOut$ statically define the range of lengths for which the algorithm can apply one of the extraction rules. If rules are K in K steps we can easily find dynamically the shortest one (that defines $trigIn$) and the longest one (that defines $trigOut$). We suppose, therefore, to have the procedures $find\_min(R)$ and $find\_max(R)$

```
      <<step>>
    }
}
```

As the $\ll initial\_step \gg$ the weight p is initialised to 0, the pointer m on the input stream in is initialised and the first symbol is read in:

```
p=0;
m=0;
car_m=read(in,m);
String curr_string=car_m;
```

where $curr\_string$ is used to contain the input elements scanned up to the current position m.
The $\ll current\_step \gg$ phase is composed of a weighted **category matching** step and, under the condition on the substring length, a **rule matching** step:

```
i=match_category(car_m);
p=p+iN^m;
len=strlength(curr_string);
if(len >= trigIn && len <= trigOut)
  delta=match_rule(p);
else
  delta=null;
```

Procedure $match\_category(car_m)$ uses array C to define the category index to which the current character belongs.
In any case, after the $\ll current\_step \gg$ phase, $delta$ can have one of the following values: $j > 0$, $null$, 0.
In the first case the algorithm executes an $\ll extract\_data \gg$ phase followed by a recursive call whereas in the other two cases the algorithm executes the $\ll step \gg$ phase.
If $delta == 0$ the algorithm executes a step-back so that it pushes $len - 1$ characters back on the input stream, updates the global pointer and performs a general reset. If $delta == null$ the algorithm has to discriminate between the cases $len < trigIn$ and $len > trigOut$. In the former case the algorithm reads one more character from in whereas in the latter case the algorithm behaves as for $delta == 0$. We note that the step-back affect computational complexity.

```
if(delta==0 || len > trigOut)
{
  in=substring(curr_string,1 , len-1)+in; \\discards the first character
  pt=pt-len+1;
  p=0; \\these three instructions perform the general reset
  m=0;
  curr_string="";
}
else \\if delta < trigIn
```

```
{
  m=m+1;
  pt=pt+1,
}
car_m:=read(in,m);
curr_string:=curr_string+car_m;
```

If *match_rule*(p) returns $delta == j > 0$ the algorithm processes the elements in *curr_string* (so to write out those that represent an instance pattern) and executes a recursive call:

```
pt=pt-delta;
sOut:=pt+'-'+substring(curr_string, len-delta, len-1)+'\n';
write(out,sOut);
extract(in, pt);
```

5.3 Something about the computational complexity

What follows is true in both the scenarios we have outlined in section 4.2. We note, however, that in the "few but complex" rules scenario, though the rules require a longer time to execute (since each of them is made of sub cases that must be sequentially checked), the execution time of every rule is independent from the length of the input data so that it can be considered constant.
As a first step we consider the **termination** of the algorithm. If n is finite, since at every recursive iteration the algorithm removes at least one element from the input stream, we have that the algorithm ends in finite number of steps.
As to the **complexity** we have that:

1. every element of the input stream must be searched for within the set of N categories each with m_i elements ($i = 0, \ldots, N-1$) and this has a cost independent from the dimension n of the input data and equal to[9] $O(l)$ where $l =\mid \Sigma \mid$.
2. the rules are loaded in an array during the initialisation phase so that the search of a rule has a cost $O(1)$.

As to the complexity of the scanning of the input stream we have that:

1. without any step-back it costs $O(n)$ in the worst case;
2. in presence of step-back we may have that, in the worst case, all the elements of *current_string* but one are inserted back in the input stream so that the shortening process of the length of **F** is: n, $n-1$, ..., 1. If we sum all such values we get $n(n+1)/2$ so that the complexity is $O(n^2)$.

[9] We note that we can do better. If we store the symbols of Σ in an bi-dimensional array of l rows and two columns, ordered according to the first column, whose generic row contains (first column) a symbol $\sigma \in \Sigma$ and (second column) the identifier of the corresponding category C_i, we can obtain the desired value with a binary search in $O(ln\, l)$ steps, again independent from n.

We note that the presence of the step-back depends on the rules but that it can hardly be avoided so that the best estimate of the complexity is $O(n^2)$ though the presence of long patterns can contribute to a faster shortening of the input stream.

6 An example: extraction of patterns from written Italian

6.1 Introduction

We define the following partition $\Sigma = C_0 \cup C_1 \cup C_2$ with, in general, $\mid C_i \mid > 2$ and consider the following cases: uniformity and non completeness, non uniformity and non completeness.
In the former case we can suppose patterns of length $len = 2$ and so, for instance, of the following types: C_0C_0, C_0C_1, C_0C_2, C_1C_1, C_1C_2, C_2C_0 and C_2C_1. We have a non balanced $N-$ary tree (non completeness) with $N = 3$ but with all the leaves at the same level (uniformity).
In the latter case we can suppose patterns of length $len \in [2,3]$ and so, for instance, of the following types: $C_0C_0C_1$, $C_0C_0C_2$, C_0C_2, C_1C_1, C_1C_2, C_2C_0, $C_2C_1C_2$ and $C_2C_1C_3$. We have a non balanced $N-$ary tree (non completeness) with $N = 3$ and with all the leaves not at the same level (non uniformity). A pattern such as C_0C_0 corresponds to an inner node and is hidden by the patterns corresponding to the leaves $C_0C_0C_1$ and $C_0C_0C_2$. As to the rules we can have only complex rules of this type:

$$R_0 = if\ curr\_string \in C_0C_0\ then\ \ll list\ of\ sub\ cases \gg \tag{6}$$

or (coding C_i as i and evaluating the current weight according to (4)):

$$if\ p == 0\ then\ \ll list\ of\ sub\ cases \gg \tag{7}$$

In this way we have $R_i \longleftrightarrow r_i \longleftrightarrow pattern_i$ if $pattern_i$ is one of the patterns and R_i is the $j-$th element of the array of the rules with $j = r_j - r_{min}$. We are going to show how all this works in the following section.

6.2 The application

In this section we show an application of the algorithm to the detection of special patterns within a stream of written Italian. In this case we have that the data of **F** either belong to the Italian alphabet Σ or to the set of punctuation symbols P or to the set of the spacing symbols S. In this case we have a partitioning:

$$F = \Sigma \cup\ P \cup\ S \tag{8}$$

with:

$$\Sigma = V \cup\ \hat{V} \cup\ C \tag{9}$$

where V are vowels, $\hat{V}$ are stressed vowels and C consonants. In this case we have $N = 5$ and we can assign to the aforesaid categories the codes 4 to V, 3

to $\hat{V}$, 2 to C, 1 to P and 0 to S. According to this convention we can have the following example of correspondence between patterns and numeric codes[10]:

1. to a pattern of the form C_0SC_1 it corresponds $r_{52} = 2*5^0+0*5^1+2*5^2 = 52$;
2. to a pattern of the form C_0SV_0 it corresponds $r_{102} = 2*5^0+0*5^1+4*5^2 = 102$;
3. to a pattern of the form $V_0V_1SV_3$ it corresponds $r_{524} = 4*5^0+4*5^1+4*5^3 = 524$.

In the first case we look for any combination of two consonants separated by one spacing symbol, in the second case we look for any combination of one consonant followed by a spacing symbol and a vowel and in the last case we look for any combination of two vowels followed by one spacing symbol and another vowel. An instance pattern of the first type is[11] *n_l* whereas one of the second type is *n_a* and one of the third is *ai_a*. With these rules we have $trigIn = 3$ and $trigOut = 4$. To coding schemes of such a type there corresponds a set of rules that are neither uniform nor complete:

1. $if\ p == 52\ then\ delta = 3$
2. $if\ p == 102\ then\ delta = 3$
3. $if\ p == 524\ then\ delta = 4$

Such rules are coded, as in all the other cases we have examined in the paper, in the $match\_rule(p)$ procedure. We note here what follows.

1. Also in case of non uniformity and non completeness we have to be careful in defining the rules so to avoid the definition of conflicting rules. In the present example, a rule associated to a pattern S_0S_1 would hide another rule associated to a pattern $S_0S_1S_2$. The same holds also for other pairs such as C_0C_1 and $C_0C_1C_2$ or $C_0C_1V_1$ and C_0C_1. From a theoretical point of view this translates in the following proposition: to avoid conflicts it is sufficient to assign rules only to the leaves of the $N-ary$ tree.
2. If we need an ability to detect finer patterns (for instance in case of C_0SC_1 if we want the rule to be applied only if $C_0 \in \tilde{C}$ and $C_1 \in \hat{C}$ with $\hat{C} \subset C$ and $\tilde{C} \subset C$) we can:
   (a) introduce sub-cases in the corresponding rule;
   (b) define a finer subdivision of C, of V or of $\hat{V}$, depending on the need.
   In the former case we can have:

$$if\ p == 52\ then\ if\ c_0 \in\ \tilde{C}\ \wedge\ c_2 \in\ \hat{C}\ delta = 3 \tag{10}$$

   where $\wedge$ means *and* and c_i is the $i-th$ character on $curr\_string$. This requires only a modification of the structure of the $categories\_file$ and the

[10] We use subscripts for readability so that C_i has the same meaning that C and the same holds for the other categories. We note that the values of the current identifier p and of the identifiers r_j are evaluated from left to right, according to the character succession of written Italian.

[11] We use the symbol _ to render the spacing symbols.

match_rule(p) procedure.
In the latter case we require that $\hat{C}$ and $\tilde{C}$ are disjoint subsets so to apply the general scheme with $N = 6$ and all the rules evaluated according the new base. This cannot be done dynamically, however, but must be foreseen when we design an implementation of the algorithm. For this reason we say that rules and categories are embedded in the code.

7 Conclusions and future plans

The proposed algorithm represents an efficient way for the extraction of a set of patterns from a stream of data. Future plans include the coding of the algorithm, its testing in real cases and the examination of the possibility of using rules and categories not embedded in the algorithm but dynamically defined.

8 Appendix: the full pseudo-code

```
R=read_rules(rules_file); \\loads the set of rules in the array
                          \\R[] from a text file rules_file
trigIn=find_min(R);
trigOut=find_max(R);
C=read_categories(categories_file); \\loads the set of categories in the
                             \\array C[] from a text file categories_file
N=size_of(C); \\evaluates the number of the categories
pt=0; \\global position of patterns within the input stream
procedure extract(in, pt)
{
     p=0;
     m=0;
     car_m=read(in,m);
     String curr_string=car_m;
     while not EOF do
     {
       i=match_category(car_m);
       p=p+iN^m;
       len=strlength(curr_string);
       if(len >= trigIn && len <= trigOut)
          delta=match_rule(p);
       else
         delta=null;
       if(delta > 0)
       {
         pt=pt-delta;
         sOut:=pt+'-'+substring(curr_string, len-delta, len-1)+'\n';
         write(out,sOut);
```

```
        extract(in, pt);
      }
      if(delta==0 || len > trigOut)
      {
        in=substring(curr_string,1 , len-1)+in; \\discards the first character
        pt=pt-len+1;
        p=0; \\these three instructions perform the general reset
        m=0;
        curr_string="";
      }
      else \\if delta < trigIn
      {
        m=m+1;
        pt=pt+1,
      }
      car_m:=read(in,m);
      curr_string:=curr_string+car_m;
    }
}
```

References

1. Lorenzo Cioni: Use of $N-$ary trees for the segmentation of a stream of data. Unpublished paper (2007)
2. Kenneth H. Rosen: Discrete Mathematics and Its Applications. WCB/McGraw-Hill (1999)
3. Adam Drozdek: Algoritmi estrutture dati in Java. Apogeo (2001)
4. Michael T. Goodrich and Roerto Tamassia: Data Structures and Algorithms in Java. John Wiley and Sons (1997)

This article was processed using the LaTeX macro package with LLNCS style