A simple algorithm for the segmentation of a stream of data

Lorenzo Cioni

Department of Computer Science, University of Pisa

e-mail: lcioni@di.unipi.it

keywords: real time processing, string processing, data segmentation, syllabification, trees

1 The nature of the problem

In this paper we face the problem of the segmentation of a stream $\mathscr{F}$ of data i.e. the insertion in such a stream of a certain number of markers in positions defined through the use of a certain number of rules $\mathscr{R}$.
The problem is solved with an ad hoc algorithm that uses the rules of the set $\mathscr{R}$. Such rules can vary depending both on the nature of the segmentation and on the type of data contained in $\mathscr{F}$. If, for instance, the data in $\mathscr{F}$ represent a stream of words written in Italian and the rules of $\mathscr{R}$ provide instructions on how to identify the syllable boundaries for written Italian, the algorithm can provide the syllabified version of such a stream ([Cio96], [Cio97]). By changing the rule set we can obtain the syllabification of any other language. In more abstract cases the algorithm allows the segmentation of $\mathscr{F}$, whose elements belong to a generic finite alphabet Σ that can be partitioned in a set $\mathscr{C}$ of disjoint categories[1] C_i.
The algorithm works in "real time" on the incoming data in $\mathscr{F}$ without requiring the full data set to be pre-loaded in any data structure for processing and so can work on input stream of any [finite] length n not ex ante known. As a result the algorithm produces an output data stream consisting of the original data and a set of segmentation markers at the suitable positions (cf.

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

Figure 1). In the present paper we use, as a segmentation marker, the character $-$ that, therefore, cannot be a legal character of the input stream $\mathscr{F}$.

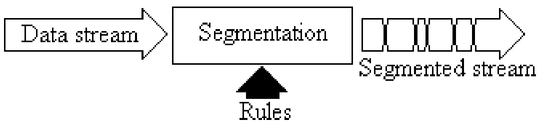


Figure 1: *Data segmentation*

2 The general structure of the algorithm

The algorithm scans $\mathscr{F}$ until a segmentation marker can be inserted. At this point the data before the insertion position are written out followed by a marker whereas those after the insertion position are appended back on $\mathscr{F}$ and a recursive call occurs. All this goes on until the EOF marker is reached on $\mathscr{F}$.

The algorithm is, therefore, composed of a main loop within which we have, in sequence: a scanning step[2] (implemented with the procedure ***scan_in(stream in)***), a category matching step (implemented with the procedure ***match_category(char car_m)***), a rule matching step (implemented with the procedure ***match_rule(int weight)***) and a certain number of conditional tests and assignment instructions. Procedure *scan_in(stream in)* scans *in* with a pointer $m \in [0, n-1]$ and loads each character or symbol[3]in a current temporary string[4] *curr_string* and in a variable *car_m*. Procedure *match_category(char car_m)* assigns each character to one of the categories C_i whose identifier i is used to evaluate a weight as a running sum on the elements of *curr_string*. Last but not least, procedure *match_rule(int weight)* uses such a weight to identify, among the rules, the one that can be applied, if it exists, on *curr_string*. Owing to the application of the rule, *curr_string* is divided in two parts: the former is written on *out* whereas the latter is written back in front of *in*. This process has some variants that will be evident in the pseudo-code.

[2]Within pseudo-code we use *in* to denote the input stream associated to $\mathscr{F}$ and *out* to denote the output stream. We also use *car_m* to denote the current symbol on the input stream whereas with σ_i we denote an element of Σ.

[3]In this paper we use the terms symbol and character as synonyms.

[4]As it will be evident from the pseudo-code in practice we need only a pointer that gives access to the elements of *curr_string*.

3 The main ingredients

The main ingredients of the algorithm are: the alphabet Σ partitioned in a set of categories $\mathscr{C}$, the set of rules $\mathscr{R}$ used to segment the data in $\mathscr{F}$ and an abstract data structure $\mathscr{T}$. Σ contains all the legal characters that can be contained in $\mathscr{F}$ and cannot include the character we choose as the segmentation marker[5]. Every element of Σ belongs to only one of the categories C_i. Strings on *in* and *curr_string* belong to Σ^*.
The rules form a set $\mathscr{R} = \{R_j \mid j = 0, \ldots, K-1\}$ where, roughly speaking, every j corresponds to a value of the running sum *weight* even if not to every such a values it corresponds a value of j and so a rule. In this case by evaluating *weight* and comparing it with the values j that indexes $\mathscr{R}$ it is possible to identify the rule to be applied or to conclude that no rule can be applied.
The abstract data structure $\mathscr{T}$ ([Ros99], [Die05]) is a generalisation of the binary tree and we call it $N-$ary tree $\mathscr{T} = (N, A)$ with[6] $\mid A \mid = \mid N \mid -1 = n-1$. We have inner nodes with direct descendants and leaves as nodes with no direct descendants. Every inner node can have the same number of direct descendants, lower than N, and we speak of a **balanced $N-$ary tree**, can have exactly N direct descendants, and we speak of a **complete $N-$ary tree**, or can have any number of direct descendants between 1 and N and we speak simply of a generic $N-$ary tree (since all category identifiers are used as least once as a label of one of the arcs). We note that completeness implies balancedness but the converse is not true: indeed we can only state that balancedness is a relaxed version of completeness.

3.1 The successions of categories

After m steps we have that *curr_string* contains m characters, each belonging to one of the C_i. We have, therefore, a succession on m instances of [repeated] categories. If we define a succession of categories[7] $C_{j_1} \ldots C_{j_m}$ as a **meaningful succession** (ms) we have that $\sigma_{j_1} \ldots \sigma_{j_m}$ (where $\sigma_{j_i} \in C_{j_i}$ for $i = 1, \ldots, m$) represents a **meaningful instances succession** (mis). Such kind of successions play a role both in the $N-$ary tree and in the segmenta-

[5]This condition is introduced mainly for practical purposes and can be easily relaxed.

[6]With $\mid A \mid$ we mean the number of elements in the set A. The symbol N is used to denote both the set of the nodes and the maximum number of the descendants of a node. Context makes clear, in every case, which is the correct meaning. In some case the cardinality of N (and so the number of nodes of a tree) will be denoted with n as the number of input data. Again, the context will make clear which is the correct meaning.

[7]We note that since we can have repeated categories not all the j_i are distinct.

tion rules since such successions can have either the same length or distinct lengths: in the first case we speak of **uniform** $m[i]s$**s** whereas in the second case we have **non uniform** $m[i]s$s.

3.2 The $N-$ary tree

In our case the $N-$ary tree $\mathscr{T}$ is characterised by the fact that every arc is labelled with exactly one category identifier C_i. Inner nodes can have any number of direct descendants depending on the nature of the tree (balanced, complete or generic) but they cannot be associated to any segmentation rule that is associated only to one of the leaves. If the leaves are all at the same level[8] we speak of a uniform $N-$ary tree: in this case leaves are associated to mss of the same length. If the tree is not uniform, leaves are associated to mss of different length. We note that an uniform tree describes mss of the same length and vice versa whereas mss of different lengths are associated to leaves at different levels and so to a non uniform tree and vice versa. Both completeness and balancedness involve inner nodes whereas uniformity concerns only the leaves so they form two groups of independent properties. The root of $\mathscr{T}$ (at level 0 and with weight 0) is associated to the empty *curr_string*, the nodes at level 1 are associated to the types[9] of the first element of *curr_string* whereas the nodes at level i are associated with the types of the $i-$th element in *curr_string*, if it is present. The outgoing arcs from every node are labelled with the categories to which the associated symbol can belong. When we reach a leaf we have identified a ms of categories to which is associated one of the segmentation rules that can be applied to *curr_string* and, then, to *in* (and correspondently to $\mathscr{F}$).

3.3 The running sum

In practice $\mathscr{T}$ is not effectively built since it can be simulated with the use of a running sum. If we denote with p the running sum or weight we have:

$$p = p + iN^j \tag{1}$$

where N is the number of categories, i is the index of the category to which *car_m* belongs, $j \in [0, M-1]$ is the position[10] of *car_m* in *curr_string* and

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

[9]We say that a character is of a type or belongs to a category C_i.

[10]With M we denote the maximum length of *curr_string* that is significant for the problem we are facing. In case of syllabification of written Italian M corresponds to the longest sequence of characters to which we associate a rule and is greater than the longest

p is put to 0 whenever *curr_string* is emptied. The value of i is obtained as $i = match\_category(car\_m)$ and requires the scan of the l elements of Σ so to define the category to which every symbol belongs[11].
In this way we evaluate all the integer values in the interval $[0, N^M - 1]$ even if, in the general case, only to some of these values it corresponds a leaf and so a segmentation rule.

3.4 The rules

The rules can be described with the following high level syntax:

$$R_j = if\ condition_j\ then\ action_j \tag{2}$$

with $j = 0, \ldots, K-1$. With $\textbf{condition}_j$ we identify an equality test between the current value of the running sum and the identifiers associated to the rules whereas with $\textbf{action}_j$ we define, possibly with the use of sub cases, essentially the position of a segmentation marker. To every rule R_j it corresponds a meaningful succession of categories C_{j_i} (with $i = 0, \ldots, I-1$ if the succession has a length equal to I) so that every rule is indeed characterised by an integer valued identifier that is evaluated as follows:

$$r_j = \sum_{m=0}^{I-1} j_m N^m \tag{3}$$

where j_m is the identifier of the symbols of Σ that belong to the category[12] C_{j_m} and that is in position m on *curr_string*. In this way we get a range $[r_{min}, r_{max}]$ to which all the K category identifiers belong. We note that we can have:

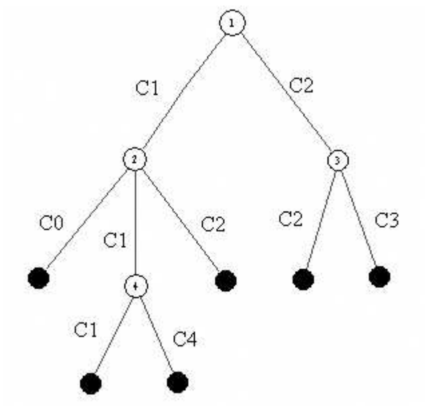
1. $K = r_{max} - r_{min} + 1$,
2. $K < r_{max} - r_{min} + 1$.

In the former case we have a **complete array of rules** with K values whereas in the latter case we have an **incomplete array of rules**. In both cases we indeed store the rules in an array $R[]$ of $r_{max} - r_{min} + 1$ elements

syllable.

[11] We can use an array of pairs *symbol ÷ category_to_which_it_belongs* so to have an access with a cost $O(l)$ in time. If such an array is ordered we can perform a binary search on it in $O(ln\, l)$ time.

[12] We remind that we are interested in mss and that to each ms is associated an integer evaluated according to relation (3). We note also that the mss are associated to leaves so that we cannot have a ms as a proper substring of another ms.



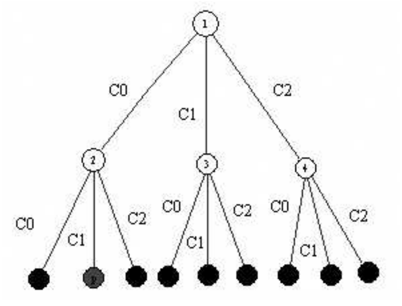


Figure 2: *Non completeness and non uniformity (left), completeness and uniformity (right)*

with an indexing scheme $id = r_j - r_{min}$ so that the search for a rule turns in a direct access to an element of an array.
In each position $R[id]$ of the array we can have either an integer value *delta* that defines the position of the marker with respect to the current position in *curr_string* or a *null* value[13] if "no rule" has been devised for that array position. In all these cases we have:

$$delta = match\_rule(p) \tag{4}$$

Within such procedure, in the simplest cases, we execute $R[p - r_{min}]$ whereas, in more complex cases, we have to access to a certain number of sub cases (that can be implemented with a switch statement or with more abstract and general structures) to which there correspond distinct values of *delta*.
Figure 2 shows (on the left) an example of **non completeness** and **non uniformity** where inner nodes are labelled as $1, 2, 3, 4$ and black dots are the leaves. In this case we have six mss (C_1C_0, $C_1C_1C_1$, $C_1C_1C_4$, C_1C_2, C_2C_2 and C_2C_3) to which there corresponds[14] six effective rules[15] on an range $[1, 106]$. On the right side of Figure 2 we have an example of a **complete** and **uniform** graph where the mss are $C_0C_0, C_0C_1, C_0C_2, C_1C_0, C_1C_1, C_1C_2, C_2C_0, C_2C_1, C_2C_2$ in the range[16] $[0, 8]$.

[13]*null* is a mnemonic for -1 and, therefore, represents an integer value.
[14]In this case we code C_i as i, with $i = 0, 1, 2, 3, 4$, and we have $N = 5$.
[15]As a general rule we have as many rules as leaves.
[16]In this case too we code C_i as i, with $i = 0, 1, 2$ whereas we have $N = 3$.

3.5 Other parameters

From the previous section it is easy to see how we have both a lower and an upper bound on the length *len* of *curr_string* on which we can try to apply one of the rules. We define such bounds respectively as *trigIn* and *trigOut*. In the case of figure 2 (left) we have $trigIn = 2$ and $trigOut = 3$. In case of figure 2 (right) we have $trigIn = trigOut = 2$.
If we have $len < trigIn$ the algorithm can only go on with scanning *in* and evaluating p since the number of symbols is insufficient whereas if we have $len > trigOut$ no rule can be applied any more so that the algorithm writes out the initial element of *curr_string* without any marker, writes back the others on *in* and goes on[17]. If $trigIn \leq len \leq trigOut$ the value of p we get at any step is used to access $R[]$ and obtain a value for *delta* to be used for subsequent computations.

4 Two toy examples and an application

4.1 Two toy examples

In the first toy example we have $\Sigma = \{a, b\}$, $C_0 = \{a\}$ and $C_1 = \{b\}$. We can have (uniformity and completeness)[18]:

1. $R_0 = if\ curr\_string\ =='' bb''\ then\ write(out,'' b-''); in ='' b'' + in$; and $delta = 1$;

2. $R_1 = if\ curr_string\ =='' ab''\ then\ write(out,'' ab-''); $ and $delta = 0$;

3. $R_2 = if\ curr_string\ =='' ba''\ then\ write(out,'' ba-''); $ and $delta = 0$;

4. $R_3 = if\ curr\_string\ =='' aa''\ then\ write(out,'' a-''); in ='' a'' + in$ and $delta = 1$;

Though the general form of a rule should be $R_0 = if\ curr\_string \in C_0C_0\ then\ write(out,'' C_0-''); in ='' C_0'' + in$; it is easy to see that this is not necessary. If we code C_1 with 1 and C_0 with 0 we have that to the above rules there correspond the following identifiers: $r_0 = r_{min} = 0$, $r_1 = 1$, $r_2 = 2$ and $r_3 = r_{max} = 3$. In this way we define $R[] = [1, 0, 0, 1]$. If on *in* we have $aabababbaa\ldots$ using the above rules we get $a - ab - ab - ba-$ on *out* and $a\ldots$ at the beginning of *in*.

[17] In this case we speak of a **step back** that has effects on the computational complexity.

[18] With "+" we define a classical concatenation operator between strings, streams and strings and the like.

As another toy example we can refer to the right side of Figure 2 and consider (without assigning any particular meaning to the symbols) $\Sigma = \{a, b, c, d, e, f, g, h, i, l\}$. We can define the partition ($C_0 = \{a, e, i\}$, $C_1 = \{b\}$, $C_2 = \{c, h\}$, $C_3 = \{d, l\}$, $C_4 = \{f, g\}$) and the set of mss we saw at the end of section 3.4. If we code C_i as i for $i = 0, \ldots, 4$ and use $N = 5$ in equation (3) we get that in the range $[1, 106]$ only 6 positions of $R[]$ (precisely positions $1, 11, 12, 17, 31, 106$) are associated to a not *null* value of *delta*. If these values, in an orderly way, are $1, 0, 1, 0, 2, 2$ this means that[19] C_1C_0 is split up as C_1- and C_0, C_1C_2 and C_2C_3 are written out as C_1C_2- and C_2C_3-, C_2C_2 is split up as C_2- and C_2, $C_1C_1C_1$ is split up as C_1C_1- and C_1 and the same holds also for $C_1C_1C_4$.

4.2 An application: the syllabification of written Italian

In this case we use the algorithm to solve a real problem. We have the alphabet Σ of written Italian that contains all the legal symbols of any written Italian text. In this case (if we use V to denote the set of unstressed vowels, $\hat{V}$ to denote the set of stressed vowels[20] and C to denote the set of consonants) we have the partition $\Sigma = V \cup \hat{V} \cup C$. In addition to these sets we have the set P of the punctuation marks and the set S of the spacing symbols. Such sets can be considered both as a part of the partition of Σ and external to it. In the former case we have $N = 5$ and the rules we devise must consider this full set of characters. In the latter case we have $N = 3$ and the rules are simpler since any occurrence of one element from either P or S brings about the direct writing on *out* of the content of *curr_string*[21]. In the present paper we follow the second approach.
At this point we have defined the alphabet Σ and the set of categories $\mathscr{C}$. We have to define a proper coding of each category C_i and the set of rules $\mathscr{R}$. We note that in this case we are in a situation like that of Figure 2, right side, and so we have neither completeness nor balancedness nor uniformity. In this paper it is not possible to give a full listing of the needed rules and we only give some examples. Further details can be found in ([Cio96], [Cio97]).

[19]In these cases the former part is written on *out* whereas the latter, if it exists, is put back at the beginning of *in*. We speak in terms of mss but similar considerations hold also for the corresponding miss.

[20]Such a set contains both wovels with an acute stress and vowels with a grave stress.

[21]We note that in cases like "fra me e te nè se nè ma" (i.e. "between me and you neither if nor but") we have non syllabification at all and spaces are, as usual, markers of word ends. The same holds for all the words of two characters and for many words of three characters that can constitute a noticeable percentage of a written text.

We therefore consider $\Sigma = \hat{V} \cup V \cup C$, $N = 3$ and the corresponding coding $2, 1, 0$ since the syllabification process (at least in Italian) is vowel driven. We remind that sets P and S (that we code respectively as 3 and 4) must anyway be known even if their elements have no role in the rules since their occurrences simply turns in the writing of the content of *curr_string* on *out* without any marker insertion.
Starting with a simple rule we have that to the *ms* $CVCV$ it corresponds the weight (or identifier[22]) 30 so that whenever the current identifier, evaluated according to relation (1), is equal to 30 we have that any combination of the form $CVCV$ is segmented as $CV-$ and CV: the first substring is written on *out* whereas the second one is inserted back at the beginning of *in*. According to this rule the Italian word *lato* (side) is syllabified as *la*$-$ and *to*. We note that we have a step back since to apply the rule we have to scan the data up to the symbol *o* but the substring *to* is put back on the input stream.
Another really more complex rule, to which it corresponds an identifier equal to 28, states that the group[23] $V_1C_1C_2V_2$ is segmented as:

1. V_1C_1- and C_2V_2 if $C_1 == C2$
2. V_1- and $C_1C_2V_2$ if $C_1 == c \mid g$ and $C_2 == h$ or if $C_1 == g$ and $C_2 == n$
3. V_1- and $C_1C_2V_2$ if $C_2 == l\ or\ r\ and\ C_1 \neq l$ and r
4. V_1C_1- and C_2V_2 if $C_1 == s\ and\ C_2 == s\ or\ V1-$ and $C_1C_2V_2$ if $C_1 = s\ and\ C_2 \neq s$
5. V_1C_1- and C_2V_2 in any other case: such open case must be handled depending on the nature of C_2.

In this case we can appreciate the complexity of a rule with all its sub cases. According to these rules we can segment words such as *assassinare* (to murder) as[24] $as - sas - si - na - re$.
We now give some indications about the cases of groups of two or more consecutive vowels. The presence[25] of group of vowels makes the syllabification harder since in Italian we can have up to six consecutive vowels (as in

[22]Weight and identifier are synonyms.

[23]We use subscripts for notational purposes, they do not affect in any way the categorisation. It is obvious that V_1 represents a variable of "type" V and assumes, as its value, one of the elements of the set $V = \{a, e, i, o, u\}$. The same holds for C_1, and the like, with respect to category C and $\hat{V}_1$, and the like, with respect to category $\hat{V}$.

[24]We show here only how the word is presented on the output stream. To each hyphen there corresponds a recursive call of the algorithm

[25]The closing part of this section is based on [Cio97].

cuoiaio, a person who sells or tans leather, to be segmented as $cuo - ia - io$) though no more than three vowels can be contained in a single syllable. The algorithm, in presence of three or more consecutive vowels, first looks for triphthongs[26] and then for diphthongs. We first introduce some group of two vowels that define a hiatus[27]. Vowels like a, e and o when are one after the other never belong to the same syllable so that *boa* (buoy) is segmented ad $bo - a$. Now we step to diphthongs. In a group like quV the vowel u form always a diphthong with V, where V, in this case, can be a, e, i or o[28]. This allows us to segment *quindi* (therefore) as $quin - di$ and *quindici* (fifteen) as $quin - di - ci$. We have diphthongs in cases such as:

1. V_1V_2 where either $V_1 = i$ and $V_2 = a, e, o, u$ or $V_1 = u$ and $V_2 = a, e, o, i$
2. V_1V_2 where either $V_2 = i$ and $V_1 = a, e, o, u$ or $V_2 = u$ and $V_1 = a, e, o, i$
3. V_1V_2 where either $V_1 = i$ and $V_2 = a, e, o, u$ or $V_1 = u$ and $V_2 = a, e, o, i$
4. V_1V_2 where either $V_2 = i$ and $V_1 = a, e, o, u$ or $V_2 = u$ and $V_1 = a, e, o, i$

According to these rules in the word[29] *cioccolato* (chocolate) we have a diphthong so that the segmented word is $cioc - co - la - to$.
The algorithm scans groups of three or more consecutive vowels from left to right and examines the first three vowels it finds. At this point the three vowels either form a triphthong, so that they must be considered a single character, or they do not form a triphthong and so the group of vowels must contain the syllable boundary.
In the first case the character that follows the triphthong can be either a vowel or a consonant. If it is a vowel, the algorithm insert the marker on its left and starts another recursive cycle. If it is a consonant, the first two vowels are written on the output stream (without any following marker) whereas the remaining vowel and the consonant are inserted back on the input stream and a recursive call occurs.
In the second case we have that the group $V_1V_2V_3$ is segmented either as $V_1-$and V_2V_3 or as $V_1V_2-$and V_3, where only one of the V_i can be stressed. If the group does not contain any stressed vowel the algorithm behaves according to the following rules:

[26]A **triphthong** is a group of three consecutive vowels that must be seen as a single character.

[27]two vowels form a hiatus if they belong to distinct syllables whereas they form a diphthong if they belong to the same syllable.

[28]In written Italian we cannot have two consecutive u.

[29]In such and similar words character i plays the role of a semi vocalic sound.

1. if $V_1 == a, o, e$ then the output is V_1- and $V_2 V_3$;
2. if $V_1 == i$ and $V_2 \neq u$ then the output is $V_1 V_2-$ and V_3;
3. if $V_1 == u$ and $V_2 \neq i$ then the output is $V_1 V_2-$ and V_3;
4. if $V_1 == i$ and $V_2 == u$ or $V_1 == u$ and $V_2 == i$ then the output is $V1$ and $V_2 V - 3$.

The first rule accounts for the segmentation of *soia* (soybean) as $so - ia$, the second for the segmentation *ghiaia* (gravel) as $ghia - ia$. Let us consider, again the hard example of *cuoiaio*. The algorithm scans the word from the left till it finds a group of three vowels *uoi* that cannot form a triphthong so it uses the proper rule and splits the triple as $uo - i$, then it goes on with scanning and find three more vowels *iai* that once again cannot belong to the same syllable and so, with another rule, it splits them as $ia - i$. At this point the algorithm finds two vowels that are known to form a diphthong and so can produce the segmented form $cuo - ia - io$. Anther example is *troiaio* (pigsty) that is segmented as $tro - ia - io$.

5 The pseudo-code

We now give the full pseudo-code of the algorithm followed by a few comments. The pseudo-code is written using a Java-like syntax for a better readability ([Dro01], [GT97]). We present the pseudo-code (with some in-line comments) in the simpler version (well suited for the toy examples) and underline where changes must be made so the adapt the algorithm to more complex and realistic applications[30].

```
// global data structures
//load the identifiers of the rules in a mono dimensional
//array
int[] R=read_rules(rules_file);
//load the categories in an array of Category
//as pairs sets_of_chars-category_identifier (int)
Category[] C=read_categories(categories_file);
int trigIn=MIN;  //lower threshold
int trigOut=MAX; //upper threshold
int r0=R[0]; //value used in the indexing scheme of R
```

[30]As to the syntax we only remind that if s is a *String* the method $s.substring(beginIndex, endIndex)$ returns the substring of s containing all the characters from the one in position *beginIndex* to the one in position $endIndex - 1$.

```
int p=0, m=0;
int base=Category.sizeOf();
String curr_string="";
char car_m='';
// main method
public void segment(in)
{
     scan_in();
     while not EOF do
     {
       i=match_category(car_m);
       p=p+i*base^m;
       len=strlength(curr_string);
       if(len >= trigIn && len <= trigOut)
         delta=match_rule(p);
       else
         delta=null;
       if(delta >= 0)
       {
         sOut=curr_string.substring(0, len-delta)+'-';
         write(out,sOut);
         in=curr_string.substring(len-delta, len)+in;
         reset();
         segment(in);
       }
       if(len>trigOut)
       {
        flush();
       }
       else
        m=m+1;
       scan_in();
     }
}
// private methods
private void reset()
{
 p=0;
 m=0;
 curr_string="";
 car_m='';
}
private void scan_in()
{
 car_m=read(in,m);
 curr_string=curr_string+car_m;
}
//performs a reset and writes out the first char of curr_string
private void flush()
{
 reset();
 sOut=curr_string.substring(0, 1);
 write(out,sOut);
 in=curr_string.substring(1, len)+in;
}
private int match_category(char car_m)
{
   for(int i=0;i<base;i++)
   {
     Category c=C[i];
     if(c.getSet().contains(char_m))
       return c.getIndex();
   }
}
private int match_rule(int p)
{
   int index=p-r0;
   return R[index];
}
```

The pseudo-code is simple and linear. Starting from the very beginning we have a general initialisation phase followed by the method *segment*(*in*). Such method performs an initial scan and then enters in a loop. Within the loop we have a **category matching** phase, some computations, a **rule matching phase** and either a segmentation and a recursive call or the flushing of *curr_string* or an advancing on *in* to the following char.
Private methods include: a **reset()** method that gives initial values to some global variables, a **scan_in()** method that reads a char from the input stream, a **flush()** method, already commented, a **match_category(char car_m)** method that is responsible for the matching between every symbol and a category and a **match_rule(int p)** method that returns the value of *delta* to be used in segmentation.

The only methods that need a variable structure are the last two. Owing to this fact we speak of rules and categories embedded in the code. In the case of syllabification (and similar ones) we have that *match_category* must manage the presence of elements of P and S, empty *currstring* and perform a general reset and a recursive call whereas *match_rule* must have an inner structure that allows the definition and the management of sub cases. In these cases we have pieces of pseudo-code like the followings:

```
private int match_category(char car_m)
{
   int id;
   for(int i=0;i<base;i++)
   {
     Category c=C[i];
     if(c.getSet().contains(char_m))
       id=c.getIndex();
       break;
   }
   if(id == 3 || id == 4)
   {
      write(out,curr_string);
      reset();
      segment(in);
   }
   return id;
}
private int match_rule(int p)
{
   int index=p-r0;
   int id=R[index];
   switch(id)
   {
     case id1:
     ...
     break;
     case id2:
     ...
     break;
     ...
     case idK:
     ...
```

```
            break;
            default:
            ...
            break;
        }
}
```

In this case (cf. section 4.2 for some examples) it is necessary indeed to associate at each value of the running sum p either a simple rule (as in the case of $p = 30$ or $CVCV$) or a complex rule with sub cases (as in the case of $p = 28$ or $VCCV$).

6 The computational complexity

As it is clear from the pseudo-code, at each step at least one element is transferred from *in* to *out*. Termination of the algorithm is therefore out of the question so let us say something about its computational complexity. Computational complexity depends strongly on the segmentation rules since rules define how many input characters are processed at any application of a rule. With this in mind, from a classical perspective we should define the best, average and worst cases.
The **worst case** occurs whenever no rule can ever be applied so at any step only one symbol is transferred form input to output stream. In this case it is easy to see that on an input stream of length n the complexity is $O(n^2)$ in time[31].
The **best case** occurs whenever we have no step back so that at any pass of the algorithm a block of symbols is transferred form *in* to *out*: in this case we have anyway to scan *in* up to the end so the complexity is $O(n)$ in time.
The **average case** depends both on the distribution of input data to be processed on Σ and on the set of the rules and the probability with which each rule is applied. As to the syllabification, such a complexity is strongly dependant on the frequencies of written Italian both from what concerns the single characters and their combinations and for what concerns the distribution of the length of the words within a "normal" written text. All this to say that, in this paper, we do not give any estimate of such a complexity

[31] We have:

$$\sum_{i=0}^{n-1}(n-i) = \sum_{j=1}^{n} j = \frac{n(n+1)}{2} \tag{5}$$

(with the substitution $n - i = j$) where i represents the number symbols transferred from *in* to *out*.

that, anyway, must fall in the range $[n, n^2]$.
As to the other operations of the algorithm we note that:

1. the operations of input scanning, assignment to car_m and $curr\_string$ occur in $O(1)$ time;
2. the operation of category matching occurs in $O(1)$ time (anyway in a time independent from n);
3. the operation of rule matching involves a direct access to an array that requires $O(1)$ time and the examination of a finite set of sub-cases that requires again $O(1)$ time (or anyway a time independent from n).

Similar considerations holds for all the other operations that are present in the pseudo-code and that have not been explicitly mentioned.

7 Concluding remarks

The proposed algorithm represents a flexible and efficient way to segment in real time a stream of data of any length under the hypotheses that the data belong to an alphabet Σ that can be partitioned in a finite set of categories and that a finite set of rules can be applied.
Possible extensions include a more flexible and less code embedded management of both the categories and the rules (and of the associated quantities) and (essentially with regard to syllabification) a better processing of prefixes based on morphological considerations.

References

[Cio96] Lorenzo Cioni. RB-tree: un algoritmo per la sillabazione dell'italiano. In *Atti del XXIV Convegno Nazionale dell'Associazione Italiana di Acustica*, volume XXIV, pages 81–84. AIA, 12-14 Giugno 1996.

[Cio97] Lorenzo Cioni. An algorithm for the syllabification of written italian. *Paper accepted at the 5th International Symposium on Social Communication, Santiago de Cuba, Cuba, January 22-24 1997 also published on Quaderni del Laboratorio di Linguistica, volume 11, Scuola Normale Superiore, Pisa, Italy*, 1997.

[Die05] Reinhard Diestel. *Graph Theory.* Springer-Verlag, 2005. Electronic Edition.

[Dro01] Adam Drozdek. *Algoritmi e strutture dati in Java.* Apogeo, 2001. Italian version of "Data Structures and Algorithms in Java", Brooks/Cole 2001.

[GT97] Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java.* John Wiley & Sons, 1997.

[Ros99] Kenneth H. Rosen. *Discrete Mathematics and Its Applications.* WCB/McGraw-Hill, 1999.