Use of $N-$ary trees for the segmentation of a stream of data

Lorenzo Cioni

Department of Computer Science, University of Pisa

e-mail: lcioni@di.unipi.it

1 Introduction

Let us suppose we have a stream of data $\mathscr{F}$ whose elements σ_i belong to N disjoint categories $\mathscr{C} = \{C_i \mid i = 0, \ldots, N-1\}$ so that we have an alphabet $\Sigma = \{\sigma_i \ : i = 1, \ldots, l\}$ with a partitioning[1]. Our aim is the segmentation of $\mathscr{F}$ as a succession of disjoint groups of elements according to an ex-ante fixed set of K rules $\mathscr{R} = \{R_j \mid j = 0, \ldots, K-1\}$ ([Cio06]). Such rules could be used also for the extraction of predefined subsets of data from the given data stream but, it the present paper, we do not examine such a possibility any further. We introduce an algorithm that gives us such a segmentation (cf. figure 1) through the use of a generalisation of the concept of binary tree, the $N-$ary tree, and we show an application to the syllabification of written Italian ([Cio96], [Cio97]).

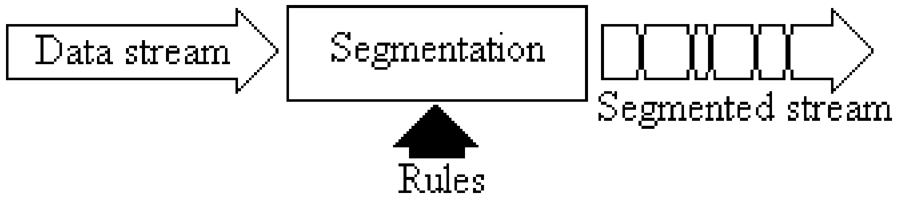

Figure 1: *Data segmentation*

[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, $l > N$.

2 Binary and $N-$ary trees

A tree ([Ros99]) 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 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** ([Ros99]), 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. A binary tree is, obviously, an $N-$ary tree with $N = 2$.

3 $N-$ary trees and segmentation

An $N-$ary tree can be used for **data segmentation** i. e. for the splitting of $\mathscr{F}$ in a stream of segments (cf. figure 1). If we identify the arcs of the tree with labels $C_i \in \mathscr{C}$ 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 string identifies a **meaningful succession** of categories to which the data to be classified through the tree belong or a succession of categories whose occurrence allows the use, on $\mathscr{F}$, of one of the segmentation rules.
Figure 2 shows an example of a part of an $N-$ary segmentation tree with the data in $\mathscr{F}$ that belong to N disjoint categories: the black dots represent inner nodes whereas the white dots represent leaves. Near to the root we have an example of a string to be segmented, be it $\gamma\sigma\alpha\beta\ldots$.
We have $\gamma \in C_i$, $\sigma \in C_{N-1}$ and $\alpha \in C_j$. Now, since during the scan of both the input string and the tree we have reached a leaf, we have identified an instance of a meaningful succession of categories to which it corresponds a segmentation rule: the input data stream $\mathscr{F}$ is therefore segmented as $\gamma\sigma\alpha$

[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.

[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.

and $\beta \ldots$. The substring $\gamma\sigma\alpha$ is written on the output data stream followed by a marker, be it $-$, whereas on the new input stream $\beta \ldots$ the algorithm is recursively applied. The algorithm stops when there are no more data in $\mathscr{F}$.

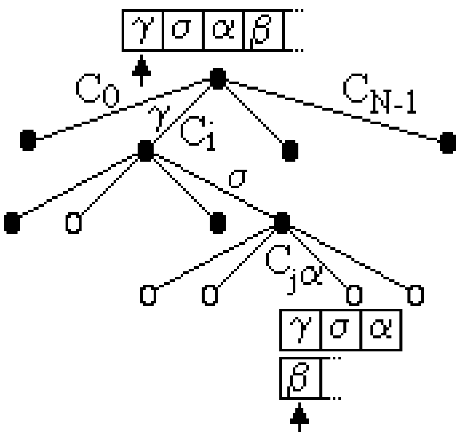

Figure 2: *N−ary trees and segmentation*

The algorithm does not really need the tree to be statically built since such a tree can be built dynamically if the input data belong to N disjoint and a priori defined categories. All 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 Segmentation rules

4.1 Introduction

A **segmentation rule** (simply **rule**) can be expressed as:

$$R_j = if\ \mathbf{condition}_j\ then\ \mathbf{action}_j\ for\ \ j = 0, \ldots, K-1 \tag{1}$$

where $\mathbf{condition}_j$ represents a logical condition that, in our context, is an equality test on substrings of $\mathscr{F}$. If the condition is verified the specified $\mathbf{action}_j$ can be applied to $\mathscr{F}$. The action turns in the segmentation of $\mathscr{F}$ through the insertion of a marker in the stream so to identify a segment of data within it as included in a pair of markers (or the beginning of the stream and a marker or a marker and the end of the stream).
Every rule R_j is univocally identified by a numeric identifier $r_j \in [r_{min}, r_{max}]$

so that **condition**$_j$ is implemented as an equality test between the current identifier 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 $K \leq \hat{k}$ 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[4].

4.2 Characterisation and properties

When we define the segmentation rules we have to face a trade−off between **quantity** and **complexity** since we can devise either many simple rules or a restricted set of complex rules. In the former case we have fine grained categories and each rule simply defines the need for the marker and its position within $\mathscr{F}$. In the latter case we have coarse grained categories and each rule contains a set of sub cases. We will briefly describe the second approach in section 6. Form here on we 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 is associated a true shift value (cf. further on). Usual cases are: completeness and uniformity, non completeness but uniformity. If we have completeness but non uniformity we have conflicting rules where longer rules (associated to leaves) hide the shorter ones (associated to inner nodes) so we consider this case as unrealistic and uninteresting. On the other hand, we are going to examine briefly the case of non completeness and non uniformity in section 6. 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 a segmentation rule (cf. section 5.1). Every action turns, therefore, in the definition of the position of the segmentation marker within $\mathscr{F}$ and in its insertion in the output data stream (cf. figure 1) together with the original data. Such a marker is an ad hoc symbol, such as $-$, that does not belong to the input data.
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. Le us suppose we have $\Sigma = \{a, b\}$ and $C_0 = \{b\}$, $C_1 = \{a\}$. If *curr_string* identifies the data on which rules act, we may have (in case of

[4] *null* is a mnemonic for -1 and, therefore, represents an integer value.

completeness and uniformity) the following rules[5]

1. $R_0 = if\ curr\_string == \text{“}bb''\ then\ write(out, \text{“}b-''); \ in := \text{“}b'' + in;$
2. $R_1 = if\ curr\_string == \text{“}ab''\ then\ write(out, \text{“}ab-'');$
3. $R_2 = if\ curr\_string == \text{“}ba''\ then\ write(out, \text{“}ba-'');$
4. $R_3 = if\ curr\_string == \text{“}aa''\ then\ write(, out\text{“}a-''); \ in := \text{“}a'' + in;$

In general we should write rules of the form:

$$R_0 = if\ curr\_string \in\ C_0C_0\ then\ write(out, \text{“}C_0-''); \ in := \text{“}C_0'' + in \quad (2)$$

but the structure of the algorithm makes this form unnecessary (cf. below). If we code C_1 with 1 and C_0 with 0 we have the following corresponding identifiers (evaluated from left to right): $r_0 = r_{min} = 0$, $r_1 = 1$, $r_2 = 2$, $r_3 = r_{max} = 3$. Moreover we have the following vector of shift values: $R[] = [1, 0, 0, 1]$. If as $\mathscr{F}$ we have $aabababbba\ldots$, we get $a-ab-ab-ab-b-ba-\ldots$. If we have uniformity but not completeness we can have:

1. $R_0 = if\ curr\_string == \text{“}bb''\ then\ write(out, \text{“}b-''); \ in := \text{“}b'' + in;$
2. $R_1 = if\ curr\_string == \text{“}aa''\ then\ write(out, \text{“}a-''); \ in := \text{“}a'' + in;$

so that we have: $R[] = [1, null, null, 1]$. If as $\mathscr{F}$ we have $aabababbba\ldots$, we get $a - ababab - b - ba\ldots$ (cf. section 5.2). In all the cases we define the integers as $trigIn$ 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.

5 The algorithm

5.1 Introduction

The algorithm accepts, as an input, a stream $\mathscr{F}$ of n elements, with n not known a priori but finite, and scans it sequentially with the aid of a pointer $m \in [0, n-1]$. The algorithm executes two steps of matching:

[5]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 stream.

1. a weighted **category matching** step that assigns every element of $\mathscr{F}$ to one of the N categories and evaluates a 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 $\mathscr{F}$.

We note that, in general, not to every weight is associated a rule. If the associated rule can be applied and the input stream has been shortened, the algorithm is recursively applied to the remaining input data of stream $\mathscr{F}$. If $car_m \in \Sigma$ is the current element on $\mathscr{F}$ (in position m) and p denotes the current weight (or current identifier) of the current substring, we have:

$$p := p + iN^m \tag{3}$$

where p is put to 0 at every recursive call and on every general reset and $i := match\_category(car_m)$ (cf. section 5.2) identifies the category to which the current element belongs. Since we have N categories and each category contains m_i elements $match\_category()$ 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 (3) 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, relative to the current position, where the segmentation marker must be inserted in order to split the input stream in two. Formally we have:

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

Under the condition $trigIn \leq m \leq trigOut$ we have that, if $delta == null$, none of the rules is associated to the current value of p otherwise, if $delta \in [0, m-1]$, one of the rules can be applied and the algorithm inserts a marker $delta$ positions before the current position. If the condition is false the algorithm either goes on with input scanning (if $m < trigIn$) or (if $m > trigOut$) writes out one element of the input string since no rule can be applied. In this last case we have a reset step and a recursive call. If $delta == 0$ there is no step-back whereas if $delta \in (0, m-1]$ we have a step-back, since only $m - delta$ elements are removed from $\mathscr{F}$ at that step.

5.2 The structure

The algorithm has the following structure[6]:

[6]We remind that *trigIn* and *trigOut* statically define the range of lengths for which the algorithm can apply the segmentation rules.

```
R:=read_rules(rules_file);
trigIn:=MIN;
trigOut:=MAX;
procedure segment(in)
{
     <<initial_step>>
     while not EOF do
     {
       <<current_step>>
       if(delta != null)
       {
         <<split_data>>
         segment(in);
       }
       <<step>>
     }
}
```

The very first statement loads, once for all, the set of rules in the array $R[]$ from a text file *rules_file* whereas the following statements give the range of lengths of the substrings of $\mathscr{F}$ for which the algorithm tries to apply the rules.
As $\ll$ *initial_step* $\gg$ we have that the pointer 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;
```

The $\ll$ *current_step* $\gg$ is composed of a weighted **category matching** phase and, under the condition of the substring length, a **rule matching** phase:

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

curr_string contains the input elements scanned up to the current position m. If $match\_rule(p)$ returns $delta == null$ we have (cf. $\ll$ *step* $\gg$) to discriminate the cases $strlength(curr\_string) < trigIn$ and $strlength(curr\_string) > trigOut$. In the latter case the algorithm writes out only one input symbol and performs a general reset.

```
if(strlength(curr_string)>trigOut)
{
  sOut:=substring(curr_string, 0, 1);
  write(out,sOut);
  in:=substring(curr_string,1, l-1)+in;
  p:=0;
  m:=0;
  curr_string="";
}
else
  m:=m+1;
car_m:=read(in(m));
curr_string:=curr_string+car_m;
```

If $match\_rule(p)$ returns $delta \,! = null$ we have a phase ($\ll$ *split_data* $\gg$) where the algorithm processes the elements in *curr_string* so to write out some of them and append back the others to the input stream:

```
 sOut:=substring(curr_string, 0, l-1-delta)+'-';
 write(out,sOut);
 in:=substring(curr_string,l-1-delta, l-1)+in;
```

We see how the $\ll$ *split_data* $\gg$ phase inserts the substring *sOut* (that includes the marker) in the output stream and the remaining substring (that can be empty) back at the beginning of the input stream *in* so that the recursive call gets the right data to process.

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 is independent from the length of the input data so that it can be considered constant (see further on).
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 $O(NM)$[7] if $M = max_{i=0,\ldots,N-1} m_i$;
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 scanning of the input stream we have that without any step-back it costs $O(n)$. If the algorithm uses a step-back we may have that, before each recursive call, a certain number of symbols is inserted back in the input stream. In the worst case all symbols but one are inserted back so the reduction process of the length of $\mathscr{F}$ is: n, $n-1$, ..., 1. If we sum all such values we get $n(n+1)/2$ so that the complexity is $O(n^2)$.

6 An example: the syllabification of written Italian

The **syllabification** of written Italian represents an example of a segmentation of a data stream where we have a set of rules that are neither complete nor uniform. In this case the elements of a stream $\mathscr{F}$ either belong to the Italian alphabet Σ or represent punctuation and spacing symbols: only the elements of the first set are processed whereas the others are simply copied from the input to the output stream. We note that $\Sigma = C \cup V \cup \hat{V}$ where V are vowels, $\hat{V}$ are stressed vowels and C consonants. Stressed vowels are the vowels with a main stress or accent $'$ on the right. In this case we have the symbols of Σ that belong to disjoint categories and a set of rules that prescribe the position of the hyphenation marker for the substrings identified by every rule.
We give here some examples of rules from [Cio97] and refer to [Cio96] and

[7]We note that we can do better. If we store the symbols of Σ in an bi-dimensional array of $S = \sum_{i=0}^{N-1} m_i$ 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 $O(lnS)$ steps.

[Cio97] for further details and a discussion of some open problems with syllabification, mainly in presence of prefixes.
We note that we have $N = 3$ and that $\Sigma = C \cup V \cup \hat{V}$ is the union of disjoint subsets. If we code the three categories of the partition of Σ respectively as 0, 1 and 2 we can define an identifier for every rule. For instance the identifier corresponding to $CVCV$ (cf. note 8) is 12. When the current identifier, evaluated according to relation (3), is equal to 12 we have that any combination of the form $CVCV$ is segmented as $CV-$ and CV: the first substring is written on the output stream whereas the second one is inserted back at the beginning of the input stream. According to this rule if, on $\mathscr{F}$, we have the Italian word *lato* (side) we apply the rule and get $la-$ 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 rule, with identifier equal to 28, states that the group[8] $V_1C_1C_2V_2$ is segmented as:

1. V_1C_1- and C_2V_2 if $C_1 == C2$
2. V_1- and $C_1C_2V_2$ if $C_2 == h$ or if $C_1 == g$ and $C_2 == n$
3. V_1- and $C_1C_2V_2$ if $C_2 == l \textit{ or } r \textit{ and } C_1 \neq l$ and r
4. V_1C_1- and C_2V_2 if $C_1 == s \textit{ and } C_2 == s \textit{ or } V1-$ and $C_1C_2V_2$ if $C_1 = s$ $\textit{and } C_2 \neq s$
5. V_1C_1- and C_2V_2 in any other case. The open case must be solved 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 (and any other we may need) we can segment words such as $assassinare$ (to murder) as[9] $as - sas - si - na - re$.
We now give some indications about the cases of groups of two or more consecutive vowels. The presence[10] of group of vowels makes the syllabification harder since in Italian we can have up to six consecutive vowels (as in *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

[8]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 V_1', and the like, with respect to category $\hat{V}$.

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

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

algorithm, in presence of three or more consecutive vowels, first looks for triphthongs[11] and then for diphthongs. We first introduce some group of two vowels that define a hiatus[12]. 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[13]. 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 *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 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 accented. If the group does not contain any accented vowel the algorithm behaves according to the following rules:

1. if $V_1 == a, o, e$ then the output is V_1- and V_2V_3 ;
2. if $V_1 == i$ and $V_2 \neq u$ then the output is V_1V_2- and V_3;
3. if $V_1 == u$ and $V_2 \neq i$ then the output is V_1V_2- and V_3;

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

[12] Vowels form a hiatus if they belong to distinct syllables whereas they form a diphthong if they belong to the same syllable.

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

4. if $V_1 == i$ and $V_2 == u$ or $V_1 == u$ and $V_2 == i$ then the output is $V1$ and $V_2V - 3$.

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 $tro - ia - io$.

7 Conclusions and future plans

The proposed algorithm represents an efficient way for the segmentation of 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.

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.

[Cio06] Lorenzo Cioni. Graphs and trees: cycles detection and stream segmentation. In *oral presentation AIRO 2006*, Cesena 12-15 September 2006.

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