ELBG Implementation

Giuseppe Patanè
Institute of Computer Science and Telecommunications
Faculty of Engineering, University of Catania
V.le A. Doria 6, 95125 Catania - ITALY,
INFN Section of Catania Corso Italia, 57, 95129 Catania - ITALY;
e-mail:gpatane@iit.unict.it

Marco Russo
Dept. of Physics - Faculty of Engineering, University of Messina,
Contrada Papardo, Salita Sperone 31, 98166 Sant'Agata (ME) - ITALY,
INFN Section of Catania Corso Italia, 57, 95129 Catania - ITALY;
e-mail: mrusso@ingegneria.unime.it,mrusso@ieee.org;
Tel: +39 (0)347 1749070

Keywords: Clustering, Unsupervised Learning, GLA, LBG, ELBG

Abstract

In this paper we describe the implementation of the ELBG, a clustering technique we developed as an improvement on the traditional LBG algorithm. It tries to solve the problem of the local minima deriving from a bad choice of the initial conditions. In a previous paper, we described in depth this technique and some points were highlighted. (*a*) It performs better than or equal to all of the other algorithms we considered. (*b*) The final result is virtually independent of the initial conditions. (*c*) No parameters have to be tuned manually (*d*) Fast convergence. (*e*) Low overhead with respect to the traditional LBG. The aim of this paper is to describe the particular solutions we adopted to obtain such results at the cost of a low overhead with respect to the traditional LBG.

I. Introduction

Vector quantization (VQ) and, more generally, unsupervised learning (or clustering), are employed in several fields. Among them, we have speech compression [1], image compression [2], pattern recognition [3] and computer vision [4].

Several approaches to clustering exist in literature, both of the fuzzy type [5], [6] and of the hard type [7], [8]. Moreover, both of these kinds of algorithms can be further subdivided in *c*-means techniques [7], [5] and competitive learning techniques [8], [9], [6]. Some authors [6], [10] say that fuzzy algorithms are less sensitive to initial conditions than hard ones. This is true if we consider the Generalized Lloyd Algorithm (GLA) [7], a hard *c*-means technique, known also as LBG (from the initials of its authors). In [11] and [12], we analyzed the reasons for such a strong dependence on the initial conditions and we proposed our solution that we called Enhanced-LBG (ELBG). The ELBG was inspired by the traditional LBG (therefore it is a hard *c*-means technique) and some improvements were made where it had weak points. In [11] and [12], the ELBG was described in depth and its performances were analyzed through several comparisons with other algorithms. Some points, in particular, were highlighted. (*a*) Its performances are better than or equal to performances obtained by all of the other algorithms we considered. (*b*) The final result is virtually independent of the initial conditions. (*c*) No parameters have to be tuned manually (many fuzzy techniques do). (*d*) Fast convergence. (*e*) Low overhead with respect to the traditional LBG.

The aim of this paper is to describe the particular solutions we adopted to keep the overhead low (below 5 %, as we said in [11]) with respect to the traditional LBG. Particular prominence is given to the tricks (regarding the logic structure of the algorithm, the data structure and the technique for accessing the data) that allowed us to obtain such a result.

The paper is organized as follows: first, some general definitions about VQ are given; afterwards, the LBG and a possible implementation of it are presented; then, we describe the ELBG and its implementation. Lastly, some results are presented.

II. Vector Quantization

A. Definition

The objective of VQ is the representation of a set of feature vectors $\mathbf{x} \in X \subseteq \Re^K$ by a set, $Y = \{\mathbf{y}_1, ..., \mathbf{y}_{N_C}\}$, of N_C reference vectors in $\Re^K$. Y is called *codebook* and its elements *codewords*. The vectors of X are called also *input patterns* or *input vectors*. So, a VQ can be represented as a function: $q : X \longrightarrow Y$. The knowledge of q permits us to obtain a partition $\mathcal{S}$ of X constituted by the N_C subsets S_i (called cells):

$$S_i = \{\mathbf{x} \in X : q(\mathbf{x}) = \mathbf{y}_i\} \quad i = 1, \ldots, N_C \tag{1}$$

B. Quantization Error (QE).

The QE is the value assumed by $d(\mathbf{x}, q(\mathbf{x}))$, where d is a generic distance operator for vectors. The mean QE (MQE), in the case that X is constituted by a finite number (N_P) of elements, is:

$$\begin{aligned} \text{MQE} \equiv D(\{Y, \mathcal{S}\}) &= \frac{1}{N_P} \sum_{p=1}^{N_P} d(\mathbf{x}_p, q(\mathbf{x}_p)) = \\ &= \frac{1}{N_P} \sum_{i=1}^{N_C} D_i \end{aligned} \tag{2}$$

where we indicate with D_i the ith cell total distortion:

$$D_i = \sum_{n:\mathbf{x}_n \in S_i} d(\mathbf{x}_n, \mathbf{y}_i) \tag{3}$$

Several functions can be adopted as distortion measures [7]. The most widely adopted is the Euclidean distance and we will use it in this paper.

III. LBG and its implementation

In this section we will briefly introduce the LBG algorithm. After, we will describe how we implemented it.

A. LBG description

LBG was proposed in 1980 [7] by Linde, Buzo and Gray and its original name was Generalized Lloyd Algorithm (GLA) because it extended the Lloyd's technique [13] from mono- to k-dimensional cases. The name LBG comes from the initials of its authors. It is an algorithm, that, at every iteration, generates a quantizer whose MQE is less or equal to the previous one. This is the result of a process where two necessary conditions for obtaining an optimal codebook are alternatively verified. They are:

- **Nearest Neighbor Condition (NNC).** Given a fixed codebook Y, the NNC consists in assigning to each input vector the nearest codeword. So, we divide the input data set in the following manner:

$$\begin{aligned} \bar{S}_i &= \{\mathbf{x} \in X : d(\mathbf{x}, \mathbf{y}_i) \leq d(\mathbf{x}, \mathbf{y}_j), \\ &\quad j = 1, ..., N_C, \, j \neq i\} \quad i = 1, ..., N_C \end{aligned} \tag{4}$$

The sets $\bar{S}_i$ just defined, constitute a partition of the input data set. This is the "Voronoi Partition" [14] and is referred to with the symbol $\mathcal{P}(Y) = \{\bar{S}_1, \cdots, \bar{S}_{N_C}\}$. It is possible to demonstrate that the Voronoi partition is optimal [7], i.e. for every partition $\mathcal{S}$ of the input data set, it holds:

$$D(\{Y, \mathcal{S}\}) \geq D(\{Y, \mathcal{P}(Y)\}) \tag{5}$$

- **Centroid Condition (CC).** Given a fixed partition $\mathcal{S}$, the CC concerns the procedure for finding the optimal codebook. This is the codebook constituted by the centroid of each cell [7].
If we consider the set $A \subset \Re^K$ constituted by N_A elements and the Euclidean distance is adopted, its centroid $\bar{\mathbf{x}}(A)$ is:

$$\bar{\mathbf{x}}(A) = \frac{1}{N_A} \sum_{\mathbf{x} \in A} \mathbf{x} \tag{6}$$

If we take the codebook $\bar{X}(\mathcal{S})$ constituted by the centroid of all the cells of $\mathcal{S}$:

$$\bar{X}(\mathcal{S}) \equiv \{\bar{\mathbf{x}}(S_i); i = i, ..., N_C\} \tag{7}$$

it is optimum [7], i.e. for every codebook Y, it holds:

$$D\{Y, \mathcal{S}\} \geq D(\{\bar{X}(\mathcal{S}), \mathcal{S}\}) \tag{8}$$

So, we can quickly identify the steps through which the LBG develops as follows:
1. initialization;
2. partition calculation according to the NNC (4);
3. termination condition check;
4. new codebook calculation according to the CC (7);
5. return to step 2.

A complete description of these steps will be given in the section concerning the LBG implementation.

B. Notation

Before we begin the description of our implementation of the LBG, we briefly explain the notation we adopted. We wrote our routines in ANSI-C and, for this reason, from now on, we will use a C-like syntax to describe many procedures. This could be a problem when we have to effect some

operations with matrix because of the several nested for loops needed to scan the data. So, the operations related to arrays and matrices will be described with a Matlab-like syntax, too.

B.1 Terminology

In the rest of the paper, we will use several matrices, arrays and scalars to store the data and we tried to give a meaningful name to each of them. Generally, their names are constituted by letters whose meaning is the following:

- P: patterns
- C: codebook (or, cell, depending from the context)
- N: number
- S: sum
- I: index
- D: distortion
- G: group
- H: hyperbox
- Un: united
- Sp: split

According to this notation, IC is the abbreviation for *Index cell*, NPC stands for *Number of Patterns in the Cell*, and so on.

Sometimes, we will substitute the expression "pattern belonging to the cell i" with the shorter term "i-pattern".

B.2 Matrices, arrays, constants and scalars

Matrices, arrays and constants are indicated in upper case, scalars in lower case. Sometimes, we wish to highlight the dimensions of a matrix or an array. In such cases, the dimensions are put in brackets, just after the name of the matrix or the array in question. For example, $C(N_C, K)$ is the matrix containing the codebook and it has N_C rows and K columns. If we specify only one dimension, than we are referring to a column array. According to this convention, the array A, constituted by N elements, $A(N)$, is equivalent to the matrix $A(N, 1)$. Later, we will also use tridimensional matrices. For example, $H(N_C, K, 2)$, is a tri-dimensional matrix whose dimensions are N_C, K, 2, respectively. Dimensions will be omitted when they are not considered to be important for improving the understanding of the context.

Single elements, rows or columns of a matrix are indicated with a Matlab-like notation. Some examples:

- $A1(2,3)[1,2]$ is the first row and second column element of the matrix $A1$. This matrix has 2 rows and three columns.
- $A1(2,3)[:,2]$ is the second column of $A1$ and $A1(2,3)[1,:]$ is the first row.

B.3 Matrix operators

Matrix operators are taken from the Matlab notation, too:

- operators for the sum and the subtraction of matrices with the same dimensions;
- operators for the product of a matrix and a scalar;
- operators for the division of a bi-dimensional matrix by an array. For example, $A2(3,2)./A3(3)$ is a matrix with 3 rows and 2 columns obtained from $A2$ and $A3$ so that its ith row is the ith row of $A2$ divided by the ith element of the column array $A3$.

B.4 Special matrices

- $zeros(r, c)$ is the matrix with r rows and c columns whose elements are all $zeros$. In a similar way, the array $zeros(r)$ is defined.
- $rand(r, c)$ is the matrix with r rows and c columns whose elements are randomly chosen. In a similar way, the array $rand(r)$ is defined.
- $infty(a, b, c)$ is the tri-dimensional matrix of dimensions a, b, c, respectively whose elements are all $+\infty$.
- $false(r)$ is the boolean array where all of the elements are $false$.

B.5 A brief recall of the C notation

- $a++$ is equivalent to $a = a + 1$;
- $a+=b$ is equivalent to $a = a + b$;
- $for(;;)$ stands for an infinite loop;
- $break$ is the condition for exiting from a loop.

C. Implementation

In our implementation of LBG, we store the N_P patterns in the matrix $P(N_P, K)$ where each row contains a learning pattern. The N_C codewords are stored in the matrix $C(N_C, K)$. Besides, we put the sum of the patterns belonging to the same cell in $S(N_C, P)$ and we use the array $NPC(N_C)$ to store the number of patterns belonging to each cell. m is the counter of the iterations and D_m is the total distortion at the mth iteration as given by eq.(2). Several techniques for calculating an initial codebook exist [7] but, in this paper, we are not interested in this phase. Therefore, we assume that a random choice of the initial codewords is enough for starting-up the algorithm. According to our symbology we can briefly describe the LBG as follows.

The LBG algorithm

// Let C_0 be the initial codebook and
// $\epsilon \geq 0$ the precision of the optimization
// process. A typical range of values for
// ϵ is $[0.001, 0.1]$.

$C = C_0$;
$D_{-1} = +\infty$;
$m = 0$;
$for(;;)$ // an infinite loop begins
{// Initialization of matrices and arrays
$S = zeros(N_C, K)$;
$NPC = zeros(N_C)$;
$D = zeros(N_C)$;
$D_m = 0$;

// Voronoi partition calculation
$for(j = 1; j <= N_P; j++)$
{i=index of the nearest codeword to $P[j,:]$
$NPC[i]++$;
$S[i,:] += P[j,:]$;
$D_m += d(P[j,:], C[i,:])$;
}

// Termination condition check
$if(\frac{D_{m-1} - D_m}{D_m} < \epsilon)$
$break$; // exit from the for loop

// New codebook calculation
$C = S./NPC$;
}

IV. The Enhanced LBG

In this section, we will summarize the considerations we made in [11] about the LBG and how they led us to the development of the ELBG. It is, essentially, a traditional LBG where a new step (the ELBG block) has been inserted between the Voronoi partition calculation and the calculation of the codebook verifying the CC. Its main function is to identify the possible situations of local minima and to remedy them. In particular, some sub-optimal solutions were adopted in order to keep the overhead introduced by the ELBG block low. Good results were achieved, experimentally, as reported in [11].

A. From LBG to ELBG

Often, the LBG converges towards a quantizer far from the optimum as the consequence of the law regulating the codewords adjustment [11]. In fact, they can "move" only through contiguous regions and, often, a bad initialization leads to a bad final quantizer because of this limitation in the codewords' movements. So, it is possible that the LBG produces a quantizer where empty cells are present, above all when large codebooks are used. However, it is easy to identify such a situation and we could, for example move the unused codeword inside a cell that is not empty, as proposed by the same authors of the LBG [7]. But, a more serious problem occurs when we are in a situation similar to the one depicted in Fig. 1(a).

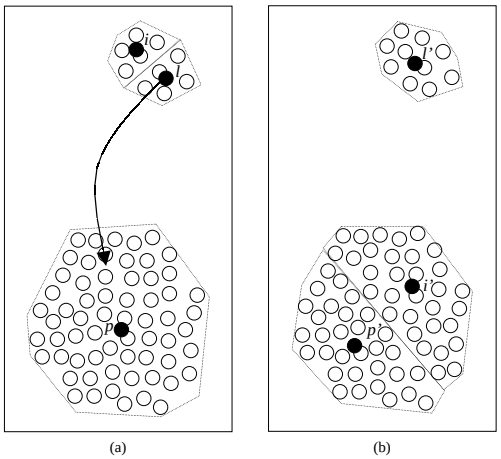

Fig. 1. (a) A situation where the codewords are badly distributed. The arrow indicates a possible solution. (b) A better distribution.

This configuration shows two clusters of patterns and three codewords. In the little cluster there are two codewords whereas, in the other, only one. For a similar distribution, the opposite situation would produce better results. But, neither the codeword i nor l, can move towards the big cluster. In [11], we developed a criterion to identify these situations and to allow the codewords to move without any contiguity limitation. The result we would like to obtain is depicted in Fig. 1(b). In the following, we will explain how such shiftings are realized. The criterion we adopted is based on the concept of "utility of a codeword (cell)" and we will explain it later on. Fritzke [15] used the term *utility* before our work, but, in our algorithm, its meaning is totally different.

B. Distortion equalization and utility

The idea of the utility was suggested to us by one of Gersho's theorems [16] where he explained his partial distortion theorem [17] saying: "*Each cell makes an equal contribution to the total distortion in optimal vector quantization with high resolution*". Gersho's theorem is true when certain conditions are verified (according to [17], a high resolution quantizer has a number of codewords tending to infinite). But, in [18], experimental results proved that it maintains a certain validity also when the codebook has a finite number of elements. So, we introduce a new step inside the LBG to pursue the equalization of the total distortions of the cells (D_i). In this context, we define the "utility index" of the ith cell as the value of D_i normalized with respect to its mean value (D_{mean}). In formal terms:

$$D_{\text{mean}} = \frac{1}{N_C} \sum_{i=1}^{N_C} D_i \qquad (9)$$

$$U_i = \frac{D_i}{D_{\text{mean}}} \qquad i = 1, ..., N_C \tag{10}$$

In the following, we will, indifferently, use the terms utility index of a cell and utility index of a codeword. Their meaning is equivalent because equations (9), (3), (10) can be used only if a cell is considered together with the related codeword and vice versa. Often, we will use only the shorter expression "utility". In terms of this new quantity, Gersho's theorem says that, in an optimal quantizer with high resolution, all the cells have utility equal to 1.

According to the definition, an empty cell has utility 0, i.e. it is useless. Besides, in Fig.1(a) both of the codewords in the little cluster have utility less than 1, while the one in the big cluster has utility more than 1.

Our idea is to obtain the desired equalization by joining a low-utility (lower than 1) cell with a cell adjacent to it. At the same time, we split a high-utility (higher than 1) cell into two smaller ones. So, it is the same as if we move the low-utility codeword inside the high-utility cell. It is possible that a better distribution, with the utilities more equalized, *could* arise from this move. The term *could* means that we are not sure that such a movement produces a real benefit. In fact, we must remember that our primary objective is the MQE minimization. So, we only make this change effective when we are sure that it produces a mean distortion decrease. In the next subsections we will describe how this evaluation is effected.

C. ELBG steps

The steps through which the ELBG develops are the same as the LBG with the addition of a new one. It is called ELBG block and is put between the termination condition check (point 3 of the LBG) and the new codebook calculation (point 4 of the LBG). Its main function is the testing of several Shift of Codewords Attempts (SoCA's). For each SoCA, we try to shift a low utility codeword inside a high utility cell, according to the considerations of the previous subsection. If this produces a decrease in the MQE, then the SoCA is confirmed and we say that a Shift of Codeword (SoC) is executed. Otherwise, the shift is discarded.

In this paper we will not deal with the initialization of the codebook. In fact, in [11], we showed with several examples that the ELBG is practically insensitive to the initial choice of codewords. Therefore, a random initialization of the codebook is sufficient to start up our algorithm. We distinguish the following steps:

1. initialization;
2. partition calculation according to the NNC (4);
3. termination condition check;
4. ELBG block;
5. new codebook calculation according to the CC (7);
6. return to step 2.

In the next subsection we will briefly summarize which operations are executed by the ELBG block.

D. The ELBG block

Inside the ELBG block, several SoCAs are executed. Each SoCA involves three cells: the first has utility lower than 1, the second is a cell adjacent to the first one and the third is a cell with utility greater than 1. The two adjacent cells are joined to form a single cell. The one with utility greater than 1 is split in two parts. If these operations produce a lower MQE, then the SoCA is confirmed and it becomes a SoC. Otherwise, it is discarded and we try a new one.

When a codeword is moved from one region towards another region, we should evaluate how this operation reflects on the MQE by the recalculation of the Voronoi partition. But, this is a time-consuming operation and we wish to avoid its execution at every SoCA. In fact, we try many SoCAs for each iteration and such a solution would be too expensive. So, we adopt a sub-optimal solution that is based on the locality of the operations. We mean that, for each SoCA, only a portion of the codewords, cells and input vectors are involved, while the remainder are not considered in any way. This is sub-optimal because a whole redistribution of the data could produce better results but, in this way, we introduce a negligible overhead to the traditional LBG and we can try many SoCAs for each iteration. However, the results are good, as we showed in [11].

The ELBG block consists in the iterated repetition of the following steps:

1. termination condition check and selection of three cells;
2. SoCA;
3. estimation of the new MQE;
4. SoC (only if MQE is lowered by the SoCA);
5. return to point 1.

D.1 Termination condition and selection of cells

To execute a SoCA we need three cells (see Fig. 1(a) for an example). They are:

- the ith cell (S_i): a cell with utility lower than 1;
- the lth cell (S_l): the cell whose codeword ($\mathbf{y}_i$) has the minimum distance from $\mathbf{y}_i$;
- the pth cell (S_p): a cell with utility greater than 1.

S_i is searched for in a sequential manner. We mean that, for the first SoCA, we start from the beginning of the codebook and, when we find a

codeword whose utility is less than one, we choose it. Afterwards, we look for the other two cells (S_l and S_p) that are needed. At the next SoCA, we continue the search for another cell S_i from the point where we stopped previously, and so on. When we reach the end of the codebook, the termination condition is verified and we try no more SoCAs for that iteration of the ELBG.

Instead, S_p is looked for in a stochastic way. The method adopted sounds like the roulette wheel selection in genetic algorithms [19]. In practice, we choose a cell with a probability P_p proportional to its utility value. In mathematical terms:

$$P_p = \frac{U_p}{\sum_{h:U_h>1} U_h} \tag{11}$$

In the section regarding implementation, we will see that the criterions we adopted for the selection of cells are a bit more complex than these. However, the description given here is enough to explain the remainder of the algorithm.

D.2 SoCA

In this step, we will describe how a SoCA is performed and we will refer to Fig. 1 as a simple bi-dimensional example. In Fig.1(a) the three cells S_i, S_l and S_p previously identified are represented. We must split the big cell S_p in two smaller cells and join S_i and S_l to form a bigger cell. This is effected by shifting $\mathbf{y}_i$ near $\mathbf{y}_p$ and executing some local rearrangements.

• **Splitting.** We said that we shift $\mathbf{y}_i$ *near* $\mathbf{y}_p$. But, what does *near* mean ? We know that a finite number of k-dimensional vectors forms the input data set. So, we can say that S_p is contained inside k-dimensional hyperbox I_p:

$$I_p = [x_{1\mathrm{m}}, x_{1\mathrm{M}}] \times [x_{2\mathrm{m}}, x_{2\mathrm{M}}] \times ... \times [x_{k\mathrm{m}}, x_{k\mathrm{M}}] \tag{12}$$

where $x_{h\mathrm{m}}$ and $x_{h\mathrm{M}}$ are respectively the minimum and maximum value assumed by the hth dimension of all the patterns belonging to S_p. From this consideration, we place both $\mathbf{y}_i$ and $\mathbf{y}_p$ on the principal diagonal of I_p; in this sense, we can say that the two codewords are *near* each other.
The exact positions are illustrated in Fig. 2. The situation of Fig. 2 can be easily generalized to a K-dimensional problem. Afterwards, a rearrangement of $\mathbf{y}_i$ and $\mathbf{y}_p$ is executed by means of a local LBG where only the patterns belonging to the old S_p constitute the input data set and the two codewords on the principal diagonal are the initial codebook. By choosing a high value for ϵ (typically $0.1 \div 0.3$), in a few iterations (one or two) the local LBG ends. In Fig. 1(b) we can see the result of this operation where the two new codewords ($\mathbf{y}'_i$, $\mathbf{y}'_p$) and the two new cells (S'_i,S'_p) are reported. This solution does not assure the best rearrangement because we did not consider in any way either the other codewords or the other input patterns. But, the locality of operations spares us the recalculation of the Voronoi partition. A lot of experimental trials have shown the validity of the method.

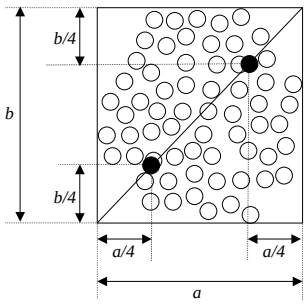

Fig. 2. The hyperbox containing S_p (I_p) and the position of the codewords on the principal diagonal of I_p.

• **Union.** After the codeword $\mathbf{y}_i$ has been moved away, we add all of the patterns belonging to the old cell S_i to S_l and we place $\mathbf{y}_l$ in the centroid of the cell so obtained. The result of this operation is reported in Fig. 1(b). In symbols:

$$\begin{cases} S'_l = S_l \cup S_i; \\ \mathbf{y}'_l = \bar{\mathbf{x}}(S'_l) \end{cases} \tag{13}$$

Also in this case we adopt a sub-optimal solution because the recalculation of the Voronoi partition for the whole input data set could produce a better result. However, our method has a much lower computational effort.

D.3 Mean Quantization Error estimation

After the shift, we have a new codebook (Y') and a new partition ($\mathcal{S}'$). Therefore, by applying eq.(2), we can calculate the new MQE. If it is lower than the value we had before the SoCA, this is confirmed, i.e. it turns into a SoC. Otherwise it is rejected. As only three cells and the related patterns are involved in the SoCA, we can effect the new MQE calculation focusing our attention only on the old three cells S_i, S_p, S_l, and the three new ones S'_i, S'_p, S'_l. The following symbols will be used:

• d_{old} is the total distortion of the three considered cells before the shift:

$$d_{old} = D_i + D_l + D_p \tag{14}$$

• d_{new} is the total distortion of the three considered cells after the shift:

$$d_{new} = D'_i + D'_l + D'_p \tag{15}$$

D.4 SoC

If $d_{new} \leq d_{old}$ we turn the SoCA into a SoC. It consists in the substitution of the new codewords ($\mathbf{y}'_i$, $\mathbf{y}'_l$, $\mathbf{y}'_p$) and the new cells (S'_i, S'_l, S'_p) in the old codebook and partition respectively. In our example, this happens by confirming the situation of Fig. 1(b). If $d_{new} > d_{old}$ the whole SoCA is discarded.

V. ELBG implementation

A more complex data structure than the one adopted to implement the traditional LBG is required to execute the operations constituting the ELBG block (point 4 of the ELBG).

First of all, for each pattern, we have to store the index of the cell to which it is assigned. This is needed because, when we want to effect a SoCA, all of the patterns belonging to the cells involved in the operation have to be identified. Such information is kept in the array $IC(N_P)$. We also have to store the utilities of all cells. However, in order to save time, we avoid the normalization according to (10) (division by D_{mean}) and we store the total distortion of each cell directly in the array $D(N_C)$. The correct execution of a splitting implies knowing which hyperbox holds the cell in question. For this reason we use the three-dimensional matrix $H(N_C, K, 2)$. In $H[c, k, 1]$ there is the smallest kth coordinate of all c-patterns. Similarly, $H[c, k, 2]$ contains the biggest kth coordinate of all c-patterns. All of the arrays and matrices just defined are filled at the same time that the Voronoi partition is calculated (point 2 of the ELBG).

A. Rearrangement of the patterns

The execution of the SoCAs (inside point 4 of the ELBG) implies a high number of accesses to the matrix of the patterns (P). Particularly, given the index of a cell, we need to locate all of the patterns belonging to it. In order to increase the efficiency of our implementation of the ELBG, we developed a method that, subject to preliminary sorting, allows us to quickly access the required elements of the matrix P.

The technique of sorting we implemented consists of the rearrangement of P so that the patterns belonging to the same cell form clusters. More precisely, we try to obtain a situation where all of the patterns belonging to the same cell are, *generally*, in subsequent rows of P. We underline the word *generally* because, as we will see later, after the execution of a SoC, the organization that we briefly described, can be slightly modified.

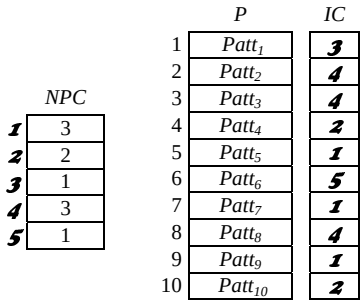

Fig. 3. An example with 10 patterns and 5 codewords. This is the situation of the matrices P, IC and NPC after the calculation of the Voronoi partition.

In Fig. 3 the situation of the patterns before sorting is depicted, practically how it is at the end of the Voronoi partition calculation. Here, and in the figures that follow, we adopt a different type of font when we refer to the indices related to patterns or to the indices related to cells. After the sorting of the data has been effected, the situation appears as in Fig. 4.

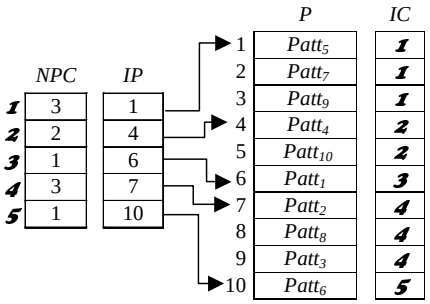

Fig. 4. The same matrices of Fig. 3 are reported after the rearrangement proposed. We can see that all of the patterns belonging to the same cell are stored in consecutive rows of P. The vector IP is reported, too.

We see how the patterns belonging to the same cell are stored in subsequent locations and that the structure is sorted according to increasing values of the field IC. In the same figure, a new array appears: $IP(N_C)$. Each element of it contains the index of the row where the patterns belonging to the cell in question begin. Such values are calculated from NPC because we know that the patterns belonging to cell 1 begin at row number 1, after there are all of the patterns of cell 2, and so on. Even if it could appear superfluous, IP allows quicker access to the data to be considered. In fact, given the general ith cell, we can immediately say that it is constituted by $NPC[i]$ patterns and that they are consecutively stored in P starting from the position $IP[i]$.

B. The technique employed for the rearrangement

As for the rearrangement of the data, we tried to reduce the number of computer memory accesses

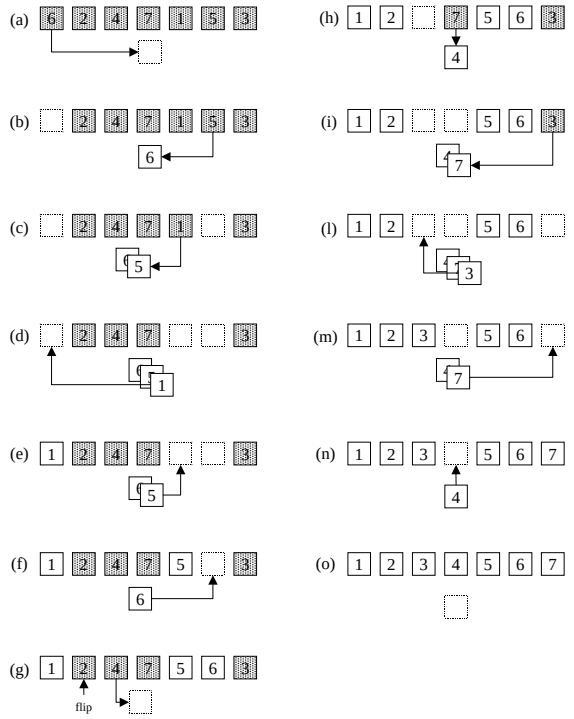

Fig. 5. Example of rearrangement.

and data movements. For this reason, we minimize the number of shiftings of the patterns by operating with their indices. We must remember that, generally, the dimensionality of the patterns is greater, or much greater, than two ($K \geq 3$). Working with indices implies that each pattern is moved once. Regarding the indices, we adopted a technique that needs a single shift for each of them. The comparisons to be effected have a linear complexity, too.

In order to simplify the exposition of our algorithm for the rearrangement, here we present a simple example illustrating the principle on which our technique is based. This example has linear complexity, too, but the number of shiftings is twice the number of elements to rearrange. The reader wishing to go into the topic is invited to consult the appendix.

Let us suppose that we have seven cards, numbered from one to seven. Let us suppose that they are lined up and hidden, as shown in Fig. 5(a). In this figure and in the following, the hidden cards are represented as shaded. We wish to reorganize them in an increasing order and we have at our disposal a temporary location where we can place the cards that cannot be put in their correct final position because it is occupied by another card. The temporary location is shown in the lower part of Fig. 5(a) as a little unnumbered square. Each time, we begin a sequence of operations from the first hidden card on the left. If it is in the right place, then we turn it over and leave it in that position. Otherwise, we put the card in the temporary location. If in this place there are also other cards, we put it onto the others. If the correct position for the current card (the one we just placed in the temporary location) is not free, we repeat the same procedure for the card that occupies that position and iterate the procedure until a free position is found. At this point, the last considered card can be correctly positioned and, if other cards are present in the temporary location, they can be put, one at a time, in their correct positions because they are free as a consequence of the method described. The procedure is repeated until no hidden cards remain and all of the cards are in the correct place.

Let us describe the complete example of Fig. 5. Sub-figure (a) shows the seven hidden cards. Let us flip over the first one and, as it is not a 1, we put it in the temporary location (a). It is a 6 and, as the sixth position is occupied, we turn the sixth card and put it in the temporary location, above the 6 (b). This card is a 5, so we turn the fifth card and put it in the temporary location, too (c). Now we can see it is a 1 and the position it should occupy is free. So, we put it in the first location (d). Now we can empty the stack of cards in the temporary location as shown in sub-figures (e)-(f). Then, let us flip over the second card and, as it is a 2, we leave it in the second position, unhidden, and go on with the third card (g). By iterating the procedure, we obtain the final result of subfigure (o) where all of the cards are sorted in an increasing order.

In the appendix we will describe in detail how, starting from this model, we implement the technique that allows us to turn the data from the situation of Fig. 3 into that of Fig. 4.

C. Access to cells whose patterns are fragmented

After the execution of a SoC, some patterns change their membership from one cell to another. In particular, when we join two cells, their patterns are, generally, stored in non-adjacent regions of the matrix P. In that case, a direct access to the patterns of the new cell is not possible if only their number and their starting position is specified. In order to avoid a global rearrangement of P after each SoC, we implemented a second type of access to the data. It is based on the use of pointers and we employ it when the patterns that we are interested in are subdivided in fragments (or groups). Each fragment is constituted by a certain number of patterns belonging to the cell in question and stored in consecutive locations. Again, the access occurs by indicating the beginning of the patterns (present in IP) and their total number (present in NPC). But, in this case, the value of IP specifies the location from where the first fragment is s-

tored. Links between fragments are managed with the help of a new vector: $IG(N_P)$. It contains two kinds of information: either the number of the consecutive patterns constituting the fragment in question, or the pointer to the first element of the next group. The two types of information are distinguished by the sign of the numeric value. An element of IG is the pointer to the next fragment when it is stored with the minus sign. The value 0 ($zero$) means that we are concerned with the last element of the last group for the cell in question.

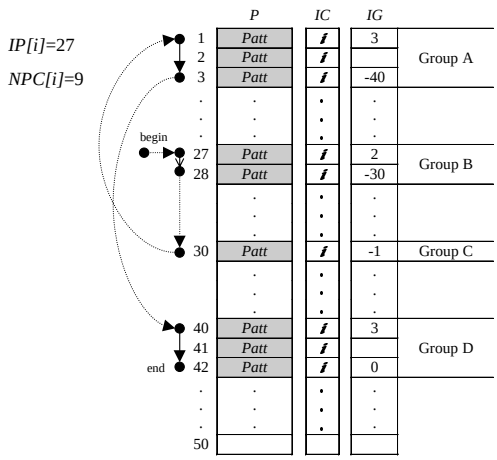

Fig. 6. Access to the patterns of the generic ith cell when they are, for example, distributed among 4 fragments.

In Fig. 6 an example illustrating such access is reported. A matrix P with 50 total patterns is represented and the 9 patterns constituting the ith cell are highlighted. Not all of the elements of IG contain meaningful information. These are exclusively in the positions corresponding to the first and the last element of each group, according to the following criteria:

- when a value of IG is related to the first element of a group, then it represents the number of patterns forming the same group;
- when a value of IG is related to the last element of a group, it is the pointer (sign-inverted) to the first location of the next fragment;
- if a group is constituted by a single element (as it is for fragment C in Fig. 6), IG holds just the index, sign-inverted, for the next fragment;
- a value of IG corresponding to the last element of the last group for a cell, holds the value 0.

According to the previous considerations, the i-patterns of Fig. 6 are visited following the order specified by the arrows, beginning from the row indicated by $IP[i]$ ($IP[i] = 27$, in this case). It should be noted that the order in which we visit the fragments depends on the links between them. Here, they are visited following the sequence B C A D.

The initialization of IG is effected after the execution of the procedure turning the data from the situation of Fig. 3 into the one of Fig. 4. In this case, IG can be obtained from NPC and IP, as the patterns are sorted according to increasing values of the field IC.

D. *Other arrays needed for the execution of the ELBG block*

As will be clear in the next sections, we need two other boolean vectors: $Sp(N_C)$ and $Un(N_C)$. The former is employed to indicate the cells arising from a splitting, the latter to indicate the cells that have joined with other cells.

In table I we report all the arrays and the matrices we use to store the data. Besides, the dimensions and a brief description are given for each of them.

Name	Dimensions	Description
P	(N_P, K)	Patterns
C	(N_C, K)	Codebook
S	(N_C, K)	Sum of the coordinates
NPC	(N_C)	Number of patterns in the cell
IC	(N_P)	Index cell
D	(N_C)	Distorsion of the cell
H	$(N_C, K, 2)$	Hyperbox
IG	(N_P)	Index group
IP	(N_C)	Index patterns
Sp	(N_C)	Split
Un	(N_C)	United

TABLE I
MATRIX AND VECTORS ADOPTED TO STORE THE DATA

VI. Description of the procedures implemented

Now, we can describe the whole algorithm. In order to make the exposition clearer, we adopt a top-down methodology. So, we start from an high-level description of the procedures and, gradually, go into details. In the practical implementation of the ELBG, every function receives a long list of parameters as input. As we do not want to make the explanation too complicated, we will avoid such listings and we will assume that all of the variables employed are global, so they are visible to all the functions.

The high-level description of the ELBG follows.

The ELBG algorithm

$C_0 = rand(N_C, K)$;
$C = C_0$;
$D_{-1} = +\infty$;
$m = 0$;
$for(;;)$ // an infinite loop begins
  {Voronoi partition calculation;
  // During the calculation of the Voronoi
  // partition, D_m is calculated, too
  $if(\frac{D_m - D_{m-1}}{D_m} <= \epsilon)$
    $break$; // end of the infinite loop
  $else$
    $\{D_{m-1} = D_m$;
    ELBG block;

// New codebook calculation satisfying CC
$C = S./NPC$,
$m++$;
}
}

Now we will detail the functions just described.

A. Voronoi partition calculation

The procedure we are about to describe is similar to that we have already seen in relation to the LBG. However, here we store a greater quantity of information with respect to the LBG.

Voronoi partition calculation

```
// Initialization of matrices and arrays
S = zeros(N_C, K);
NPC = zeros(N_C);
D = zeros(N_C);
D_m = 0;
H[:,:,1] = +infty(N_C, K, 1);
H[:,:,2] = -infty(N_C, K, 1);

// Identification of the cells and calculation
// of the related information
for(j = 1; j <= N_P; j++)
  {i= index of the nearest codeword to P[j,:];
  S[i,:]+ = P[j,:];
  NPC[i]++;
  D[i]+ = d(P[j,:], C[i,:]);
  D_n+ = d(P[j,:], C[i,:]);
  for(r = 1; r <= K; r++)
    {H[i,r,1] = min(H[i,r,0], P[j,r]);
    H[i,r,2] = max(H[i,r,1], P[j,r]);
    }
  IC[j] = i;
  }
```

B. ELBG block

In this subsection we will describe our implementation of the ELBG block. The schematic description of the procedure follows; after, we will explain some of its particulars.

ELBG block

```
// First of all, the patterns are rearranged from
// the unsorted form of Fig. 3 to the sorted
// form of Fig. 4
global sorting of the data;
//Initialization of Sp, Un and IG
Sp = false[N_C];  Un = false[N_C];
IG is initialized as explained in V-C;

for(i = 1; i <= N_C; i++)
  {D_mean = D_m/N_C;
  if((D[i] < D_mean)  AND  (Sp[i] == false))
  {// Let us begin the selection of the
  // cells needed for a SoCA
  if(NPC[i] == 0
    {l = 0;
    // such a value means that looking for
    // the cell S_l is not necessary because
    // the cell S_i is empty
    }
  else
    {// look for the cell S_l
    l=index of the nearest codeword to C[i,:];
    if(Sp[l] == true)
    // We assign to l a value indicating that
    // S_l has been previously split (and
    // now it cannot be joined to another cell)
      l = -1;
    }
  if(l >= 0)
    {// Let us look for p (S_p is the cell
    // to be split)
    p = Roulette_wheel();
    if(p > 0) // S_p was found
      {SoCA;
      // During the SoCA, d_old and d_new
      // are also calculated, according to
      // (14) and (15), respectively
      if(d_new < d_old)
        {SoC;
         D_m+ = d_new - d_old;
        }
      }
    else break // exit from the for loop
    }
  }
}
```

After the rearrangement of the data and some operations related to the initializaton, the array containing the distorsions of the cells is scanned sequentially. When a low-utility cell S_i is found (i.e. $D[i] < D_{\text{mean}}$), we look for the other two cells (S_l and S_p) needed to effect a SoCA. Let us remember that S_i should be joined to S_l, while S_p should be split into two cells. However, some considerations allow us to smartly reduce the number of SoCAs effected at each iteration. They are:

- we do not allow a cell coming from one splitting (or more) to be joined to other cells, even if its utility is lower than 1. In fact, if a cell derives from the splitting of another one because it was too big, we think it is not suitable trying to expand it again. For this reason, such cells are identified by setting as *true* the corresponding value in the boolean array Sp;
- we do not allow a cell coming from one previous union (or more) to be split, even if its utility is higher than 1. In fact, if two (or more) cells had been joined to form a bigger one, a splitting could

create the previous situation again. The cells deriving from unions are identified by the value *true* in the boolean array Un.

These two considerations help us to simplify the execution of the SoCAs, as will be explained later. Therefore, before proceeding with a SoCA, we verify if the three cells S_i, S_l and S_p satisfy the requirements. If S_i or S_l do not, we go on with the sequential scanning of the codebook looking for another cell S_i from which a new SoCA could begin. Instead, if no valid cell S_p is found, then the ELBG block ends. In fact, this means that no more cells with utility lower than 1, and not previously united, exist. The search for S_l (and the whole procedure of the union of S_i with S_l) is bypassed if S_i is empty. In fact, in such a situation, it is not necessary to assign any pattern to another cell when $\mathbf{y}_i$ is moved away.

B.1 Looking for the cell S_p

S_p is searched by the roulette-wheel method (11). Now we will describe the complete procedure to select S_p; we must remember that codewords deriving from unions cannot be split.

Roulette_wheel()

```
// Let us calculate and store in x the sum
// of the distorsions of all the cells that can
// be split.
for(x = 0, p = 1; p <= N_C; p++)
  {if(D[p] > D_mean  AND  Un[p] == false)
    x+ = D[p];
  }

// Let us verify that at least one cell with utility
// greater than 1 and not deriving from previous
// unions has been found.
if(x == 0)
  {// The procedure ends and the value p = 0
  // is returned.
  return  p = 0;
  }

// Let us find the cell S_p with the stochastic law
// described by (11). First, let us
// generate a uniformly distributed
// random number y in the range [0, x]
y = random(x);
for(x = 0, p = 1; p <= N_C; p++)
  {if(D[p] > D_mean  AND  Un[p] == false)
    {x+ = D[p];
    if(x >= y) // this value of p is returned
      return  p // the procedure ends
    }
  }
```

B.2 Description of a SoCA

Now, we will describe a SoCA. Most of the operations executed store their results in auxiliary locations of the memory. So, if the SoCA is confirmed (i.e., it turns into a SoC), the values are copied to the general locations, otherwise they are discarded. The names of the auxiliary arrays and matrices are the same as the general locations. However, they are distinguished by means of primes. Particularly, we indicate with a prime the arrays and the matrices employed for the splitting and with two primes the ones we use for the union. According to this convention, $C'(2, K)$ is the matrix that will store the codewords ($\mathbf{y}'_i$ and $\mathbf{y}'_p$) deriving from the splitting of S_p. $C''(1, K)$ will hold the codeword ($\mathbf{y}'_l$) coming from the union of S_i and S_l, and so on. Some results are stored just in the general locations because, even if the SoCA should be discarded, any wrong information they hold would be ignored. So, the correct continuation of the algorithm would not be compromised.

	NPC	IP	Sp	Un
l	5	6	*false*	
p	10	20		*false*
i	3	34	*false*	

	P	IC	IG
1			
.	.	.	.
.	.	.	.
.	.	.	.
6	$Patt_6$	*l*	3
7	$Patt_7$	*l*	
8	$Patt_8$	*l*	-43
.	.	.	.
.	.	.	.
.	.	.	.
20	$Patt_{20}$	*p*	10
21	$Patt_{21}$	*p*	
22	$Patt_{22}$	*p*	
23	$Patt_{23}$	*p*	
24	$Patt_{24}$	*p*	
25	$Patt_{25}$	*p*	
26	$Patt_{26}$	*p*	
27	$Patt_{27}$	*p*	
28	$Patt_{28}$	*p*	
29	$Patt_{29}$	*p*	0
.	.	.	.
.	.	.	.
.	.	.	.
34	$Patt_{34}$	*i*	3
35	$Patt_{35}$	*i*	
36	$Patt_{36}$	*i*	0
.	.	.	.
.	.	.	.
.	.	.	.
43	$Patt_{43}$	*l*	2
44	$Patt_{44}$	*l*	0
.	.	.	.
.	.	.	.
.	.	.	.
50			

Fig. 7. Situation of the data related to the cells i, l, p, before the SoCA.

Once S_i, S_l and S_p have been found, we have the situation of Fig. 7. The values of Sp and Un for the three cells involved are also reported; they are:

- S_i: $Sp[i] = false$ because S_i must be joined to S_l and cannot come from previous splittings. We are not interested in the value of $Un[i]$.

• S_l: is the same as S_i.
• S_p: $Un[p] = false$ because a cell to be split cannot come from previous unions. We are not interested in the value of $Sp[p]$.

In the following, we report the scheme for the SoCA and, after, we explain it.

Description of a SoCA

// The three cells S_i,S_l,S_p are fixed (the
// last one is needed only if S_i is not empty).

// Operations related to splitting.
// The results are stored in $C'(2,K)$, $S'(2,K)$,
// $NPC'(2)$, $D'(2)$, $H'(2,K,2)$, $IC'(2)$.
splitting of S_p in S'_i and S'_p;

// Operations related to the union of
// S_i and S_l in S'_l.
// The results are stored in $C''(1,K)$,
// $S''(1,K)$, $NPC''(1)$, $D''(1)$.
$if(NPC[i] > 0)$ // S_i is not empty
  $\{S''[1,:] = S[i,:] + S[l,:]$;
  $NPC''[1,:] = NPC[i,:] + NPC[l,:]$;
  $C''[1,:] = S''./NPC''$;
  // The distortion of the cell S'_l is calculated
  $D''[1] = \sum_{P[r,:] \in S_i \cup S_l} d(P[r,:], C''[1,:])$
  // Let us store in fip the index of the
  // first i-pattern; it will be needed if
  // the SoCA becomes a SoC to link the
  // pattern of the two cells in the
  // data structure.
  fip=index of the first i-pattern;
  }
// Calculation of the distortion related to
// the old three cells (S_i, S_l, S_p) and
// to the new ones (S'_i, S'_l, S'_p).
$if(NPC[i] > 0)$ // the cell i is not empty
  $\{d_{old} = D[i] + D[l] + D[p]$;
  $d_{new} = D'[1] + D'[2] + D''[1]$;
  }
$else$ // the cell i is empty
  $\{d_{old} = D[p]$;
  $d_{new} = D[i'] + D[p']$;
  // Of course, in this case it will be
  // $d_{new} <= d_{old}$
  }

• *Operations related to the splitting.* In the previous scheme, we neglected all of the details related to the splitting because the procedures to apply have already been described in IV-D.2. Besides, as P has been rearranged, the input patterns we are interested in are stored in consecutive rows of P. So, the two procedures to be applied for the local LBG (calculation of the Voronoi partition and calculation of the codebook satisfying the CC) are almost identical to the ones applied to the whole data structure of the ELBG. The access to the portion of the data involved in the operation occurs by specifying the position from where all of the p-patterns are stored and their number. These values are stored in $IP[p]$ and $NPC[p]$, respectively (Fig. 7). In IV-D.2, we explained how the initialization of the codebook for the local LBG is effected and we said that the hyperbox holding S_p must be known. This information is stored in $H[p,:,:]$.
• *Operations related to the union.* Let us remember that this phase is not executed if S_i is empty because, in this case, the removal of the codeword related to it would not leave any pattern to assign to other cells. In addition to the calculation of the centroid for the new cell S'_l, we keep in the memory the index of the last i-pattern. So, if the SoCA turns into a SoC, we already have the value needed to link the two cells in the data structure, too. This operation is shown in Fig. 8.

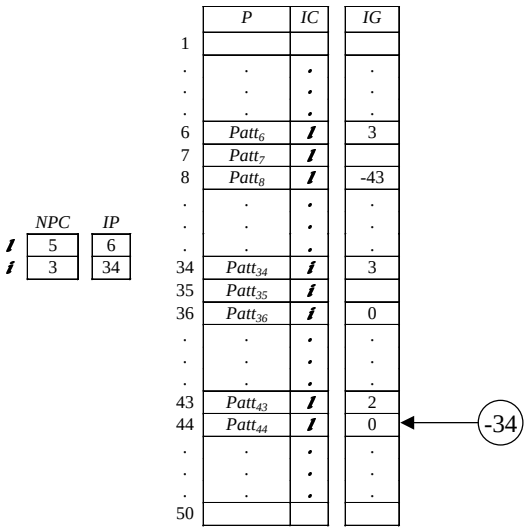

Fig. 8. Linking of the last l-pattern to the first i-pattern. This operation will be executed during the SoC.

B.3 Description of a SoC

When the distortion we obtain by substituting the old three cells and codewords (S_i, S_l, S_p and the related codewords) with the three new ones (S'_i, S'_l, S'_p and the related codewords) is lower than before the SoCA, this is confirmed and the corresponding SoC is executed. In practice, it consists in the copying of the results of the SoCA from the auxiliary locations of the memory to the general ones. Besides, the data structure must be adjusted so that the access can continue to occur as described in V-A and V-C. The schematic description of the SoC is reported here; after we will explain it.

Description of a SoC

// Operations related to the union.
// If the old cell S_i is empty, no operation
// related to the union is executed.
$if(NPC[i] > 0)$ // the cell S_i is not empty
{// Copy of data to general locations
$C[l,:] = C''[1,:]$;
$S[l,:] = S''[1,:]$;
$NPC[l,:] = NPC''[1,:]$;
$D[l,:] = D''[1]$;
// S_l is identified as deriving from a union
$U[l] = true$;
// Linking of the patterns belonging to
// the two cells. The value of fip
// (first i-pattern) was stored
// during the SoCA
llp=index of the last l-pattern;
$IG[llp] = fip$;
}

// Operations related to the splitting.
// Copy of data to general locations.
$C[p,:] = C'[1,:]$; $C[i,:] = C'[2,:]$;
$S[p,:] = S'[1,:]$; $S[i,:] = S'[2,:]$;
$H[p,:,:] = H'[1,:,:]$; $H[i,:,:] = H'[2,:,:]$;
$NPC[p] = NPC'[1]$; $NPC[i] = NPC'[2]$;
$D[p] = D'[1]$; $D[i] = D'[2]$;
local sorting of the patterns related to S'_i and S'_p;

// Adjustment of the other vectors.
$IP[i] = IP[p] + NPC[p]$;
// $IP[p]$ does not have to be modified
// S_i and S_p have to be identified as deriving
// from a splitting and not from a union.
$Sp[i] = true$; $Sp[p] = true$;
$Un[i] = false$;
// $Un[p]$ was already $false$

• *Operations related to the union.* The first operations concern the copying of the data from the auxiliary to the general locations. Moreover, S_l, that has grown because patterns have been added, has to be identified as deriving from a union by setting $Un[l] = true$. Afterwards, the linking between the patterns of the two cells that have joined is effected as shown in Fig. 8. Instead, IC and H are not modified because their values are necessary only when a splitting occurs. But, cells deriving from unions cannot be split.

• *Operations related to the splitting.* After the data have been copied from auxiliary to general locations, S_i and S_p are identified as coming from splitting by setting $Sp[i]$ and $Sp[p]$ to the value $true$. Moreover, it is necessary to set the value of $Un[i]$, too. In fact, before the splitting, $Un[i]$ was not considered in any way. Now, the old cell S_i no longer exists (while S_l has grown because the old i-patterns have been added) and has been substituted by one of the two cells coming from the splitting. So, according to the previous considerations, it can be further split and this is possible only if we set $Un[i] = false$.
Before the splitting, the p-patterns were stored in consecutive locations of memory. After the splitting, this is not true any longer, as we can see in the example of Fig. 9. However, we can restore the order by executing a simple local rearrangement. This means that only the data related to the split cell are involved. In fact, all of them are stored in consecutive locations and the same procedure we apply to the whole data structure can be applied only to the region in question. So, data can still be accessed with the techniques previously described. It is not necessary to modify IG because its values are used only when we have to join two cells. But, cells deriving from a splitting cannot be joined to other cells.

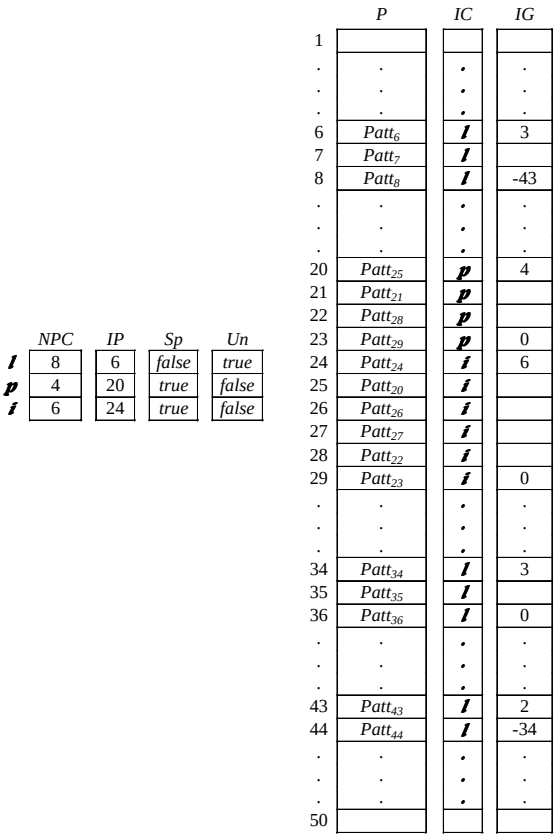

Fig. 10. Situation of the data related to the cells i, l, p, after the SoCA

After the SoC, we have the situation of Fig. 10 and it has to be compared with that of Fig. 7.

VII. Results and comparisons

In this section we will report some comparisons concerning the compression of the famous image of Lena [20]. For a complete set of comparisons, the reader is invited to look at [11].

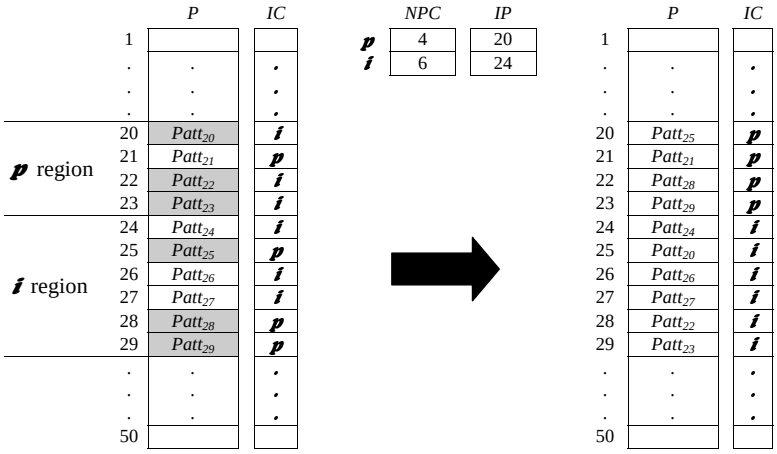

Fig. 9. Local rearrangement of the patterns.

In image compression, a widely adopted measure for evaluating the process is the Peak Signal to Noise Ratio (PSNR). For an 8-bit grey level image, it is defined as:

$$\mathrm{PSNR} = 10 \log_{10} \frac{255^2}{\frac{1}{IJ} \sum_{i=0}^{I-1} \sum_{j=0}^{J-1} (f(i,j) - \hat{f}(i,j))^2} \tag{16}$$

where $f(i,j)$ and $\hat{f}(i,j)$ are respectively the grey level of the original image and the reconstructed one. All grey levels are represented with an integer value comprised in $[0, 255]$.

The 8-bit grey level Lena's image of 512×512 pixels was divided into 4×4 blocks and the resulting 16384 16-dimensional vectors were used as input data set. We compare the results produced by the ELBG with those of Lee et al. [21]. They presented an enhanced performance K-means algorithm which improved both the classical K-mean algorithm (the LBG) and Jancey's method [22]. The results are summarized in Table II. Our tests are averaged on 5 runs. In this case we improved both the error and the number of required iterations. In [11], we, experimentally, demonstrated that, in this situation, the ELBG block introduces an increase in the required time per iteration that is maintained below 5%.

N_C	Modified K-means		ELBG	
	PSNR (dB)	Iter.	PSNR (dB)	Iter.
256	31.92	20	31.94	10.4
512	33.09	17	33.14	10.6
1024	34.42	19	34.59	11.8

TABLE II

LEE ET AL. AND ELBG COMPARISON

Another comparison is reported with the work of Karayiannis and Pai [10]. In this case, the 8-bit grey levels Lena's image of size 256×256 is used and a codebook of 512 codewords is constructed. As their method depends on several parameters, they executed several runs with different parameter values. The best result they obtained for PSNR was a value of 32.62 dB. With a codebook randomly initialized we obtained a PSNR of 33.04 dB as the average of 5 runs.

APPENDIX

Now we will describe the procedure we execute to lead the data from the unsorted form of Fig. 3 to the sorted situation of Fig. 4. This technique derives from the one described in V-B and illustrated in Fig. 5. There, we showed how the sorting of a certain number of cards (7, in that example) developed through sequences of operations. We tried to optimize such a procedure to reduce the number of shiftings through the locations of memory that the patterns are subjected to. This is realized by means of a stack of indices helping us to identify the order of the shiftings to effect before their actual realization. So, inside a sequence of operations, we are able to directly move all of the patterns (except for one) from the starting location to the correct one without their having to pass through auxiliary positions. Vectors that are already in their correct locations are never moved. The technique of Fig. 5 was based on the *a priori* knowledge of the final position that each card had to occupy. Here, the situation is slightly different because each pattern can, correctly, be placed inside a range of positions, not just one. To make the concept clearer let us consider, as an example, Fig. 3. There, an input data set of 10 patterns belonging to 5 different cells is represented. From this figure, let us construct Fig. 11.

Here, we subdivided P into 5 regions, one for each cell. This was done assuming that, when the matrix will be sorted, the patterns belonging to cell 1

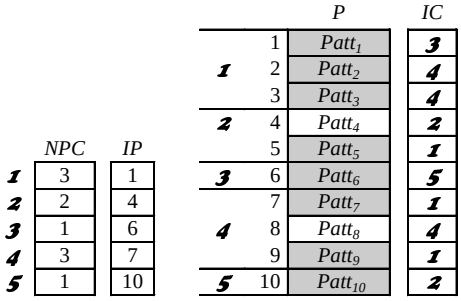

Fig. 11. Initial situation. Only the arrays and the matrices involved in the sorting operation are reported.

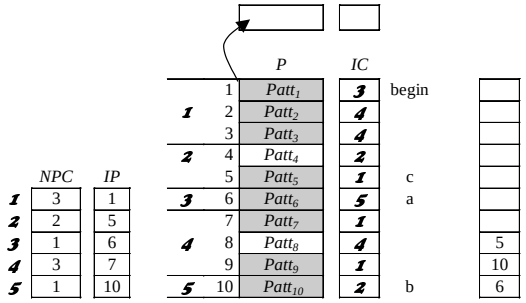

Fig. 12. Sorting: stack creation.

start from row 1 of P, then the patterns belonging to cell 2 follow, and so on. In this way, the generic i-pattern, can, correctly, occupy any position inside region i. In Fig. 11 and in the following pictures, unshaded rows represent the patterns that already occupy a correct position, i.e. those for which the value of IC is equal to the number of the region where they are. Shaded rows identify the patterns that have to be shifted. The opportunity of shifting vectors inside a range rather than to a single position, gives us more freedom. However, it is necessary to establish a rule that allows us to quickly identify the location that the pattern will have to occupy. For this reason we use the array IP. According to the definition given in V, in a configuration like that of Fig. 4, IP holds the indices of the rows where each region begins. During the sorting, IP plays a different role. Before the procedure starts, it identifies the beginning of the regions, as we can see in Fig. 11. Afterwards, it is opportunely adjusted so that the generic element $IP[i]$ contains the index of the first row of region i that is not occupied by an i-pattern. So, when, during the rearrangement, we run into an i-pattern, it is assigned to the row $IP[i]$ (even if the shifting will occur later). Then, $IP[i]$ is updated so that it contains the index of the next row in region i not occupied by an i-pattern. Working like this, patterns already positioned in a correct region are never involved in the rearrangement. When all of the rows in region i have been assigned, $IP[i]$ assumes a value outside the range of region i. But, it is no longer meaningful because we will not run into other i-patterns to arrange.

In Fig. 12 we begin to illustrate the procedure. First of all, let us notice that $IP[2] = 5$ has been set because row 4, the first in region 2, is already occupied by a pattern that can remain in that position. Now, we will list the steps through which the stack of the indices related to the first sequence of operation is created. In this phase no pattern is moved; only the order to follow for the shiftings is determined, as we can see in Fig. 12.

- Let us begin from the first vector not occupying a correct position ($P[1,:]$, in this case), let us shift it into the auxiliary location and let us keep in the memory the number of the region where the initial position has been freed; here, it is region number 1. Now, we have to identify a succession of vectors to shift until we find one belonging to the region we started from. It is necessary to complete the sequence.
- $P[1,:]$ has to be moved into region 3, in the row specified by $IP[3]$ ($IP[3] = 6$). Let us put the value 6 in the stack, increase by one $IP[3]$ and consider $P[6,:]$.
- $P[6,:]$ belongs to cell 5, so it has to be shifted to the row indicated by $IP[5]$ ($IP[5] = 10$). Let us put the value 10 in the stack, increase by one $IP[5]$ and consider $P[10,:]$.
- as $P[10,:]$ belongs to cell 2, it has to be moved into the row indicated by $IP[2]$ ($IP[2] = 5$). Let us put the value 5 in the stack, increase by one $IP[2]$ and consider $P[5,:]$.
- $P[5,:]$ belongs to cell 1; so it is the element that allows us to complete the first sequence of operations because it can be shifted into the region we started from. At this point we are with the situation reported in Fig. 13.

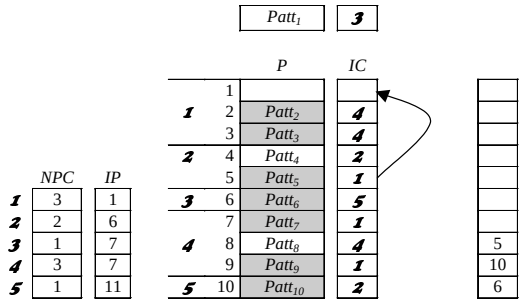

Fig. 13. Emptying of the stack (I).

Now we are ready to begin the shiftings of the patterns following the order determined by the stack, that we will empty according to the rule Last In First Out (LIFO). The element (it is an index) on the top of the stack is taken and the pattern corresponding to that index is put in the empty row of P. Then, we remove that element from the

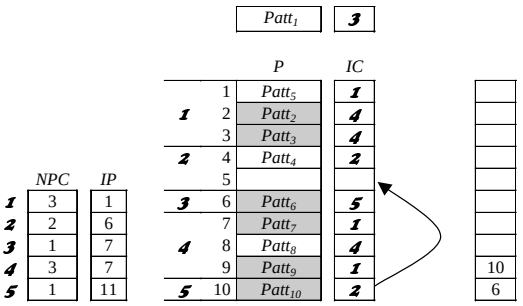

Fig. 14. Emptying of the stack (II).

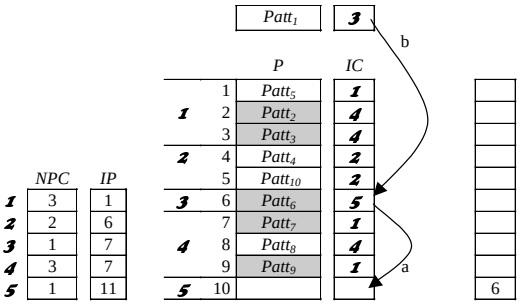

Fig. 15. Emptying of the stack (III).

stack and, iteratively, repeat this procedure until the stack is empty. (Fig. 13-15). Afterwards, the pattern in the temporary location is shifted into the empty row of P (Fig. 15) and $IP[1]$ is increased by one.

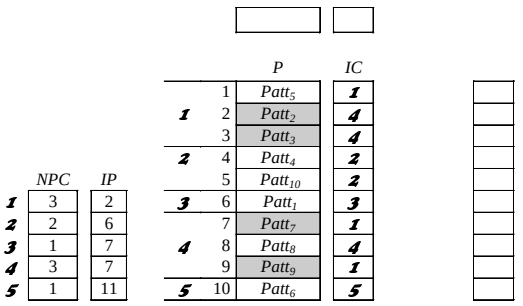

Fig. 16. Situation after the completion of the first sequence of operations.

After the first sequence of operations, we have a situation as in Fig. 16. The procedure just described starts again from row 2 and continues until all the vector is sorted as we wish.

With this technique, for each sequence of operations, all of the patterns (except the one the sequence begins from) are directly shifted from the initial to the final location. We could work as in Fig. 5, by considering P and IC as a single matrix and placing the record pattern-cell into the stack. But, in that case, all of the patterns involved in the sequence of operation are shifted twice.

When all the data have been sorted, the right values for IP are restored so that they identify the beginning of each region. The final situation is that of Fig. 4.

References

[1] K.K.Paliwal and B.S.Atal, "Efficient Vector Quantization of LPC Parameters at 24 Bits/Frame," *IEEE Transactions Speech And Audio Processing*, vol. 1, no. 1, pp. 3–14, 1993.

[2] P.C.Cosman, R.M.Gray, and M.Vetterli, "Vector Quantization of Image Subbands: A Survey," *IEEE Transactions on Image Processing*, vol. 5, no. 2, p-p. 202–225, 1996.

[3] K.Fukunaga, *Introduction to Statistical Pattern Recognition*. 24–28 Oval Road, London NW1 7DX: Academic Press Limited, second ed., 1990.

[4] J.M.Jolion, P.Meer, and S.Bataouche, "Robust Clustering with Applications in Computer Vision," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 13, pp. 791–802, Aug. 1991.

[5] J.C.Bezdek, "Pattern Recognition with Fuzzy Objective Function Algorithms," in *New York: Plenum*, 1981.

[6] N.B.Karayiannis and P.-I Pai, "Fuzzy Vector Quantization Algorithms and Their Application in Image Processing," *IEEE Transactions on Image Processing*, vol. 4, pp. 1193–1201, 1995.

[7] Y.Linde, A.Buzo, and R.M.Gray, "An Algorithm for Vector Quantizer Design," *IEEE Transaction on Communications*, vol. 28, pp. 84–94, Jan. 1980.

[8] T. Kohonen, *Self organization and associative memory*. Berlin: Springer Verlag, 3rd ed., 1989.

[9] N.R.Pal, J.C.Bezdek, and E.C.K.Tsao, "Generalized Clustering Networks and Kohonen's Self Organizing Scheme," *IEEE Transaction on Neural Networks*, vol. 4, pp. 549–557, July 1993.

[10] N.B.Karayiannis and Pin-I Pai, "Fuzzy Algorithms for Learning Vector Quantization," *IEEE Transaction on Neural Networks*, vol. 7, pp. 1196–1211, Sept. 1996.

[11] M.Russo and G.Patanè, "The Enhanced-LBG Algorithm," *IEEE Transactions on Knowledge and Data Engineering*, submitted.

[12] M.Russo and G.Patanè, "Improving the LBG Algorithm," in *Proc. of IWANN'99* (J.Mira and J.V.Sánchez-Andrés, eds.), vol. 1606 of *Lecture Notes in Computer Science*, (Barcelona, Spain), pp. 621–630, Springer, June 1999.

[13] S.P.Lloyd, "Least Squares Quantization in PCM's." Bell Telephone Laboratories Paper, Murray Hill, 1957.

[14] A.Gersho and R.M.Gray, *Vector Quantization and Signal Compression*. Boston: Kluwer, 1992.

[15] B.Fritzke, "The LBG-U Method for Vector Quantization – an Improvement Over LBG Inspired from Neural Network," *Neural Processing Letters*, vol. 5, no. 1, pp. 35–45, 1997.

[16] A.Gersho, *Digital Communications*, ch. Vector Quantization: A New Direction in Source Coding. North-Holland: Elsevier Science Publisher, 1986.

[17] A.Gersho, "Asymptotically Optimal Block Quantization," *IEEE Transaction Information Theory*, vol. IT-25, no. 4, pp. 373–380, 1979.

[18] C.Chinrungrueng and C.H. Séquin, "Optimal adaptive K-Means Algorithm with Dynamic Adjustament of Learning Rate," *IEEE Transaction on Neural Networks*, vol. 6, pp. 157–169, Jan. 1995.

[19] M.Russo, "FuGeNeSys: A Genetic Neural System for Fuzzy Modeling," *IEEE Transactions on Fuzzy Systems*, vol. 6, pp. 373–388, Aug. 1998.

[20] D.C.Munson, Jr., "A Note on Lena," *IEEE Transactions on Image Processing*, vol. 5, p. 3, Jan. 1996.

[21] D.Lee, S.Baek, and K.Sung, "Modified K-means Algorithm for Vector Quantizer Design," *IEEE Signal Processing Letters*, vol. 4, pp. 2–4, Jan. 1997.

[22] M.R.Anderberg, *Cluster Analysis for Applications*. New York: Academic, 1973.