Unione Europea
Fondo Sociale Europeo

Ministero dell'Università e della
Ricerca Scientifica e Tecnologica

Università degli studi di
Palermo

Dottorato di Ricerca in Ingegneria Elettronica,
Informatica e delle Telecomunicazioni
XIII Ciclo

Unsupervised Learning on Traditional and Distributed Systems

Autore della tesi:
Giuseppe Patanè

Tutor:
Chiar.mo Prof. Ing. Salvatore Cavalieri

Coordinatore del corso:
Chiar.mo Prof. Ing. Giovanni Mamola

Tesi cofinanziata dal Fondo Sociale Europeo
PROGRAMMA OPERATIVO MULTIREGIONALE 1994/99
n. 940023 I 1
"Ricerca, Sviluppo Tecnologico ed Alta Formazione" Misura 1.1.
sottomisura Dottorati di Ricerca

Unsupervised Learning on Traditional and Distributed Systems

Giuseppe Patanè [1]

[1]e-mail:gpatane@ai.unime.it, gpatane@libero.it

Contents

1 Introduction **9**
1.1 Objective of the thesis 9
1.2 Clustering and Vector Quantization 10
1.3 Traditional algorithms for CA and VQ 12
1.4 Incremental algorithms for CA and VQ 13
1.5 Parallel Algorithms for CA and VQ 14
1.6 Organization of the thesis 15

2 Vector Quantization **16**
2.1 Definition 16
2.2 Evaluation of the quantization error 16
2.2.1 Source whose statistical properties are known 17
2.2.2 Source whose statistical properties are unknown 18
2.3 Distortion measure 18
2.3.1 Mean quantization error measures 19
2.4 Optimal quantizer 20
2.4.1 Nearest Neighbor Condition 20
2.4.2 Centroid Condition 20

3 Basic algorithms for hard and fuzzy CA/VQ **22**
3.1 Generalized Lloyd Algorithm (GLA) or LBG 22
3.1.1 Codebook optimization 23
3.1.2 Initialization of the codebook 25
3.2 Considerations about the LBG algorithm 26
3.3 Fuzzy techniques 27
3.4 Considerations about hard and fuzzy techniques 28

4 The Enhanced LBG Algorithm **30**
4.1 Introduction 30
4.2 General considerations 31
4.3 Distortion equalization and Utility 33

4.4 Detailed description of the ELBG block 34
4.4.1 Termination condition 35
4.4.2 Selection of cells 36
4.4.3 Codeword shift and local rearrangements 36
4.4.4 Mean Quantization Error estimation 39
4.4.5 Confirmation or discarding of the SoCA 41
4.5 Considerations on the utility concept 41
4.6 ELBG overhead estimation 44
4.7 Results and comparisons 45
4.7.1 Qauntization of bi-dimensional cases 45
4.7.2 Image compression 50
4.8 Conclusions 53

5 The Enhanced LBG Implementation 55
5.1 Notation 55
5.1.1 Terminology 55
5.1.2 Matrices, arrays, constants and scalars 56
5.1.3 Matrix operators 57
5.1.4 Special matrices 57
5.1.5 A brief recall of the C notation 57
5.2 LBG implementation 57
5.3 ELBG implementation 59
5.3.1 Rearrangement of the patterns 59
5.3.2 The technique employed for the rearrangement 60
5.3.3 Access to cells whose patterns are fragmented 62
5.3.4 Other arrays needed for the execution of the ELBG block 64
5.4 Description of the procedures implemented 64
5.4.1 Voronoi partition calculation 66
5.4.2 ELBG block 66
5.4.3 Looking for cell S_p 68
5.4.4 Description of a SoCA 69
5.4.5 Description of a SoC 72
5.5 Detailed description of the technique employed for the rearrangement 75

6 Fully Automatic Clustering System (FACS) 81
6.1 Introduction 81
6.2 Considerations regarding ELBG 82
6.3 General Description 82
6.4 Smart Growing 84

6.4.1 Discussion about the law regulating the decrease of the target error . . . 87

6.5 Smart Reduction . . . 88

6.6 Behaviour of FACS versus the number of iterations and termination condition . . . 90

6.7 Discussion about outliers . . . 91

6.8 Results and comparisons . . . 93

6.8.1 Introduction . . . 93

6.8.2 Comparison with ELBG . . . 94

6.8.3 Comparison with GNG and GNG-U . . . 97

6.8.4 Comparison with FOSART . . . 99

6.8.5 Comparison with the Competitive Agglomeration Algorithm . . . 99

6.8.6 Classification . . . 100

6.9 Conclusions . . . 105

7 The MULTISOFT Machine 107

7.1 Introduction . . . 107

7.2 Hardware organization . . . 107

7.3 Management . . . 109

7.4 Updating the system . . . 112

8 Parallel Implementation of LBG and ELBG 113

8.1 Introduction . . . 113

8.2 Parallelization of LBG and ELBG . . . 113

8.2.1 PARLBG . . . 114

8.2.2 PARELBG . . . 118

8.3 Results . . . 121

8.4 Conclusions and future developments . . . 122

9 PAUL: A Parallel Algorithm for Unsupervised Learning 123

9.1 Introduction . . . 123

9.2 The algorithm . . . 124

9.2.1 General considerations . . . 124

9.2.2 The essential points of the new algorithm . . . 126

9.2.3 Determining the portions . . . 128

9.2.4 Full procedure and inter-process communication . . . 130

9.2.5 Memory management . . . 132

9.3 Results and comparisons with previous works . . . 134

9.3.1 Compression of large-sized images . . . 134

9.3.2 Comparison with PARLBG and PARELBG . . . 137

9.3.3 Texture segmentation 139
9.4 Conclusions and future works 142

10 Conclusions and further studies 143

11 Acknowledgments 145

List of Figures

1.1 Clustering and Vector Quantization. (a) original data set; (b) the pair of clusters identified by the CA algorithm; (c) VQ with 4 codewords; (d) VQ with 8 codewords. 10

3.1 The LBG procedure 23
3.2 LBG codebook optimization 24
3.3 Splitting of a codeword 26
3.4 Badly positioned centroids 26

4.1 ELBG codebook optimization 32
4.2 High-level flow-chart of the ELBG block 33
4.3 Detailed description of the ELBG block 35
4.4 Initial situation before the SoCA 36
4.5 The cell S_p and the hyperbox containing it 37
4.6 Codewords position immediately after the shift 38
4.7 Codewords position and patterns distribution after the local rearrangements 38
4.8 The initial distribution of the utility indexes 41
4.9 The final distribution of the utility indexes when the LBG algorithm is used 42
4.10 The final distribution of the utility indexes when the ELBG algorithm is used 42
4.11 σ_l versus the number of iterations in the LBG case 43
4.12 σ_r verus the number of iterations in the LBG case 43
4.13 σ_l verus the number of iterations in the ELBG case 43
4.14 σ_r verus the number of iterations in the ELBG case 43
4.15 Confidence intervals of the percentage time increase per iteration with a confidence level of 99% 44
4.16 RMSE versus number of iterations. ELBG$_{\mathrm{rnd}}$ (solid line) and LBG$_{\mathrm{spl}}$ (dashed line) 46
4.17 Initial distribution 47
4.18 LBG final distribution 48

4.19 ELBG final distribution 48
4.20 LBG final distribution 48
4.21 ELBG data set and codewords adjustment 49
4.22 Original image of Lena 51
4.23 LBG reconstructed image of Lena 51
4.24 ELBG reconstructed image of Lena 52
4.25 PSNR in dB versus the size of the codebook 53

5.1 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. 60
5.2 The same matrices of Fig. 5.1 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. 60
5.3 Example of rearrangement. 61
5.4 Access to the patterns of the generic ith cell when they are, for example, distributed among 4 fragments. 63
5.5 Situation of the data related to the cells i, l, p, before the SoCA. 70
5.6 Linking of the last l-pattern to the first i-pattern. This operation will be executed during the SoC. 73
5.7 Local rearrangement of the patterns. 75
5.8 Situation of the data related to the cells i, l, p, after the SoCA 76
5.9 Initial situation. Only the arrays and the matrices involved in the sorting operation are reported. 77
5.10 Sorting: stack creation. 78
5.11 Emptying of the stack (I). 79
5.12 Emptying of the stack (II). 79
5.13 Emptying of the stack (III). 79
5.14 Situation after the completion of the first sequence of operations. 80

6.1 A FACS-iteration 83
6.2 The two phases through which FACS develops 83
6.3 The *Smart Growing* phase 85
6.4 p versus the number of iterations 86
6.5 Detailed description of the insertion of the codewords 87
6.6 The *Smart Reduction* phase 88
6.7 Detailed description of the deletion of the codewords 89
6.8 Typical trend of N'_C (it is N_C normalized with respect to its value after 200 iterations) and D (normalized with respect to e_T) versus the number of iterations 90

6.9 Dataset with two clusters and one outlier 91
6.10 Prototypes of the Competitive Agglomeration Algorithm after 10 iterations 100
6.11 Codewords of FACS after 6 iterations 101
6.12 Two spirals: supervised classification. $e_T = 0.01$; $N_C = 74$ . . 103
6.13 Two spirals: supervised classification. $e_T = 0.001$; $N_C = 144$. 104
6.14 Two spirals: unsupervised classification. $e_T = 0.001$; $N_C = 143$ 105

7.1 The structural of the MULTISOFT machine 108
7.2 The FSM describing the OS organization of the MULTISOFT machine 110

8.1 Task vs. Time diagram for PARLBG with $k = 16$, $N_P = 16384$, $N_C = 128$, $N = 4$. The scale of times is such that an iteration is about 0.75 s. 115
8.2 Task vs. Time diagram for PARLBG with $k = 16$, $N_P = 16384$, $N_C = 256$, $N = 16$. The scale of times is such that an iteration is about 0.60 s. 116
8.3 Task vs. Time diagram for PARELBG with $k = 16$, $N_P = 16384$, $N_C = 128$, $N = 4$. The scale of times is such that an iteration is is about 1.27 s. 120

9.1 Operations executed by the generic task 127
9.2 RMSE versus iteration number for ELBG (solid line) and PAUL for $N = 20$ (dotted line) 137
9.3 Estimate of the speed up for N_C=256 138

List of Tables

4.1 Execution times in seconds per iteration 44
4.2 Number of required iterations 45
4.3 $x_2 = 8x_1^3 - 3x_1$ 46
4.4 Cantor's distribution 47
4.5 ELBG and LBG-U comparison 50
4.6 Lee et al. and ELBG comparison 50

5.1 Matrix and vectors adopted to store the data 65

6.1 ELBG performance 95
6.2 FACS performances 96
6.3 GNG performances 98
6.4 FOSART comparison 99

8.1 Speed up (S) for PARLBG and PARELBG 119

9.1 Performance of PAUL with *giraffe* 135
9.2 Maximum estimated speed ups vs the number of codewords . 138
9.3 Comparison of the results related to PARLBG, PARELBG and PAUL 139

Chapter 1

Introduction

1.1 Objective of the thesis

The aim of this thesis is the presentation of the work and the results related to the study and the research performed during the triennium of my Ph.D.

In this context, several new algorithms for Cluster Analysis (CA, or clustering) and Vector Quantization (VQ) [1–8], developed during this period, will be shown.

In some cases, the techniques considered have to deal with the elaboration of large amounts of data. In such circumstances, in order to increase the available computing power, the utilization of techniques of parallel computing can be an effective solution. For this reason, with the cooperation of Prof. Marco Russo of the University of Messina (Italy), we have also worked in the construction of the MULTISOFT Machine, a powerful farm of Personal Computers employed for the implementation of several algorithms. The equipments constituting the MULTISOFT Machine have been mainly funded by the *Istituto Nazionale di Fisica Nucleare (INFN)* - section of Catania and also by *Istituto Nazionale di Fisica della Materia (INFM)* - section of Messina and *Centro di Calcolo Elettronico - Università di Messina (CECUM).*

In this introductory chapter, a brief description of the main themes faced will be given. It begins with the definition of the objectives of CA and VQ and the identification of several field of application. Afterwards, the new techniques developed and presented in the thesis are classified inside one of three families of algorithms for CA/VQ (traditional, incremental and parallel). Such techniques are ELBG (belonging to the family of traditional algorithms), FACS (among the incremental algorithms), PARLBG, PARELBG and PAUL (in the family of parallel algorithms). Lastly, the organization of the following chapters is described.

1.2 Clustering and Vector Quantization

CA is an important instrument in engineering and other scientific disciplines. Its applications cover several fields such as pattern recognition [9–11], texture and image segmentation [12–14], boundary detection and surface approximation [15], magnetic resonance imaging [16,17], handwritten character recognition [18], computer vision [14,19], information retrieval [20,21], data mining [22] and machine learning [23].

According to Jain et al. [24], CA is the organization of a collection of patterns (usually represented as a vector of measurements, or a point in multidimensional space) into clusters based on similarity. Intuitively, patterns within a valid cluster are more similar to each other than they are to a pattern belonging to a different cluster. Pattern proximity is usually measured by a distance function defined on pairs of patterns.

In many applications regarding telecommunictions and other fields where the compression and/or the transmission of data is involved, several techniques based on algorithms for VQ are often used [25–30]. Also in this case, patterns are subdivided into groups (or *cells*), based on similarity measured by a distance function [31–39]. Each cell is represented by a vector (called *codeword*) approximating all of its elements. The set of the codewords is called the *codebook*.

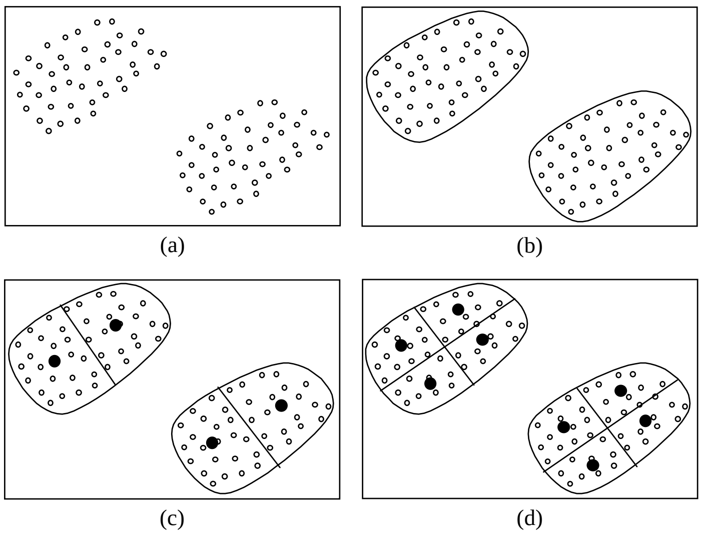


Figure 1.1: Clustering and Vector Quantization. (a) original data set; (b) the pair of clusters identified by the CA algorithm; (c) VQ with 4 codewords; (d) VQ with 8 codewords.

In the wide scientific community using techniques for CA and/or VQ, a widespread opinion is that they are different. CA is, generally, conceived as the problem of identifying, with an unsupervised approach, the eventual clusters inside the multi-dimensional data set to be analyzed [9,40–44]. Differently, a VQ algorithm is not intended on finding the clusters, but on representing the data by a reduced number of elements that approximate the original data set as well as possible. The concept is clearer with the help of some figures. Let us consider, for example, the bi-dimensional patterns in Fig.1.1(a). The aim of a CA algorithm is, generally, to automatically identify the two clusters present, as shown in Fig. 1.1(b). A good algorithm can also autonomously recognize the number of the clusters, even if it is not known *apriori* and also in the presence of noise/outlier points. [14,45]. While, if we apply a VQ technique to the same data set, the number of cells into which the data have to be subdivided depends only on the desired degree of approximation. The higher the number of the codewords (and likewise of the cells), the better the degree of approximation. Four and eight cells (with the related codewords) are highlighted in Figs 1.1(c) and 1.1(d), respectively. It is evident that, in the latter case, the degree of approximation is better than in the former.

In spite of the differences outlined here, it is possible to demonstrate that, in many cases, CA and VQ are, practically, equivalent [46–48]. Summarizing, we can say that, often, from an operative point of view, the two approaches roughly execute the same operations: grouping data into a certain number of groups so that a loss (or error) function is minimized.

In the remainder of the thesis, we will use a symbology and terminology typical of VQ. Besides, we will use also the term *Unsupervised Learning* (UL) to indicate one or both of the two approaches.

After the introduction of a unified terminology for identifying both techniques for CA and VQ, we present a distinction between two main cathegories of algorithms for UL: they can be, generally, classified as hard (crisp) [32] or soft (fuzzy) [49, 50] techniques. The difference between these is mainly the degree of membership of each vector to the clusters. During the construction of the codebook, in the case of the hard group, each vector belongs to only one cluster with a unitary degree of membership, whereas, for the fuzzy group, each vector can belong to several clusters with different degrees of membership. All of the new algorithms presented in the thesis belong to the hard family. However, at the end of chapter 3, the formulation for a typical fuzzy problem is given, too. Besides, a brief comparison between the two approaches is given.

1.3 Traditional algorithms for CA and VQ

With this name we define that family of iterative algorithms for CA/VQ where, iteration by iteration, the values of the parameters to be optimized vary, while their number is unchanged.

Inside this family, we can distinguish two main groups: K–means and competitive learning. The clustering techniques belonging to the first scheme try to minimize the average distortion through a suitable choice of codewords. In the second case the codewords are obtained as a consequence of a competition process between them. Generalized Lloyd Algorithm (GLA), sometimes called the Linde-Buzo-Gray (LBG) algorithm [32], belongs to the first group. Recent developments in Neural Networks (NNs) architectures resulted in several competitive learning algorithms [51] as, for example, the well known Learning Vector Quantization (LVQ) and the Self-Organizing Feature Map (SOFM) [52]. Other competitive learning clustering techniques are Fuzzy LVQ (FLVQ) [49], Fuzzy Algorithms for LVQ (FALVQ) [53, 54] Generalized LVQ (GLVQ) [55], and GLVQ Fuzzy (GLVQ-F) [56].

The performance of several VQ algorithms depends on the choice of the initial conditions and the configuration parameters. Pal et al. in [55] analyzed the behaviour of the LVQ algorithm. They showed that a bad codebook initialization implies very bad results. They made an attempt to resolve this problem proposing GLVQ. But, in [57], Gonzalez et al. demonstrated that the validity of GLVQ is restricted to a small domain of applications. They showed that the performance of GLVQ can drastically deteriorate with a uniform resizing of the data set. Successively, Karayiannis et al. [56] resolved this drawback with a fuzzy modified version of GLVQ they called GLVQ-F. Karayiannis and Pai underline that fuzzy techniques are less sensitive than others [53, 58]. FALVQ [53] is effectively less sensitive, but, unfortunately, it depends strongly on the right choice of several parameters required by the algorithm itself. The same thing happens with the Fuzzy c-means (FCM) technique [49]. Among hard techniques, some based on competitive learning succeed in overcoming the previous problems. Chinrungrueng et al. [59] proposed a new technique based on an optimal adaptive algorithm that is less sensitive to the initial codebook and to the necessary parameters of the algorithm itself. Using stochastic relaxation such as simulated annealing, some authors proposed other types of VQ algorithm to get good codebooks independently of the initial codewords [36].

In chapter 3, LBG [32], one of the most famous algorithms in literature, will be described in detail. Practically, it is equivalent to the traditional hard K-means clustering algorithm [46, 60] and is the basis for the new techniques proposed in this thesis. The first of such techniques is the Enhanced LBG

(ELBG), that will be presented in chapter 4. ELBG is designed to solve the main problems affecting LBG and its main characteristics are:

- performances better than or equal to performances obtained by all of the other algorithms considered;
- the final result is virtually independent of the initial conditions;
- no parameters have to be tuned manually (many fuzzy techniques do);
- fast convergence;
- low overhead with respect to LBG.

In chapter 5, the particular solutions adopted to keep the overhead low (below 5 %,with respect to LBG) are presented. 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.

1.4 Incremental algorithms for CA and VQ

In literature, several techniques for VQ/CA exist where, as the algorithm develops, not only the values of the parameters to be optimized vary, but also their number. For this reason, we define them incremental techniques. They are used for problems of both Supervised Learning (SL) and Unsupervised Learning (UL). Among them, we cite: Growing and Splitting Elastic Nets (UL) [61], Incremental Radial Basis Functions Networks (SL) [62], Growing Cell Structures (GCS, SL and UL) [63], Growing Neural Gas (GNG, SL and UL) [64, 65] that take Neural Gas [66] (UL) as a starting point, Fuzzy ARTMAP (SL) [67] that derives from the Adaptive Resonance Theory (ART) of Grossberg [68], FOSART [48, 69] and the Competitive Agglomeration Algorithms [14, 45].

In chapter 6, the Fully Automatic Clustering System (FACS) is presented. It is a VQ/CA iterative algorithm whose aim is, given a data set and a target error e_T, to find a codebook that approximates the input data set with an error less than e_T. The cardinality of the codebook, such as the codewords themselves, is a parameter that the system, in a completely automatic way, identifies during its execution. At each iteration, FACS tries to improve the setting of the existing codewords and, if necessary, some elements are removed from or added to the codebook. In order to save on the number of computations per iteration, *greedy* techniques, i.e. techniques of local updating, are adopted. It is also demonstrated, from a heuristic point of view, that the number of the codewords is very low and that the algorithm quickly converges towards the final solution.

1.5 Parallel Algorithms for CA and VQ

In several CA/VQ applications, very complex problems, with a high number of patterns and codewords, have to be considered. In such cases, both the calculation time and the memory occupation can be a problem. For this reason, the scientific community has tried to develop parallel algorithms for UL that can benefit from the use of parallel or distributed resources as regards both the computational power and the availability of physical memory.

Various hardware architectures have been employed such as, for example: specialized architectures [70], massively parallel processors [71], transputers [72, 73] and networks of workstations [6, 74–78].

Because of the low cost involved in its realization, a solution that is spreading very fast consists of clusters of Personal Computers (PCs) using low-cost and high-availability hardware; they are also called commodity supercomputers. In spite of their low cost, such systems can be very useful instruments in scientific computing. In fact, some of them are in the top-500 ranking of the most powerful supercomputers in the world [79]. This is a very interesting prospective and, for this reason, also the IEEE has devoted a section to such systems [80].

For implementing the new parallel CA/VQ techniques proposed in this thesis, we built and used one of such systems, i.e. the MULTISOFT Machine, a powerful farm of Personal Computers whose description, together with the description of the particular techniques adopted for its administration, are described in chatper 7.

As regards the parallel CA/VQ algorithms developed, a preliminary study about such techniques is reported in chapter 8. It consists of the parallel implementation of LBG and ELBG, called PARLBG and PARELBG, respectively. The aim of this preliminary study is to individuate which parts of LBG and ELBG can easily be implemented on a system like the MULTISOFT Machine and which ones have to be modified for improving the efficiency. Starting from the results collected from PARLBG and PARELBG, a new parallel algorithm has been developed. It is named PAUL and is presented in chapter 9. PAUL derives from ELBG but some important modifications have been performed in order to allow it to be efficiently implemented on the MULTISOFT Machine. The results reported show that the modifications introduced let PAUL obtain good results as regards both the final quantization error and the speed up with respect to ELBG.

1.6 Organization of the thesis

The thesis is organized as follows

- in Chapter 2 some basic definitions about VQ are given;
- in Chapter 3 LBG and FCM are presented; besides, a brief comparison between hard and fuzzy techniques is given.
- in Chapter 4 ELBG and the related results are presented;
- in Chapter 5 the particular tricks adopted for implementing efficiently ELBG are presented;
- in chapter 6 FACS and the related results are presented;
- in chapter 7 the MULTISOFT Machine and the particular techniques adopted for its adiministration are described;
- in chapter 8 PARLBG, PARELBG and the related results are presented;
- in chapter 9 PAUL and the related results are presented;
- **Manca ancora il capitolo delle conclusioni**

Chapter 2

Vector Quantization

In this chapter, some important definitions about hard VQ are given; besides, some necessary conditions for a quantizer to be said optimum, are given.

2.1 Definition

The objective of VQ is the representation of a set of feature vectors $\mathbf{x} \in X \subseteq D^k$ by a set, $Y = \{\mathbf{y}_{\mathit{1}}, ..., \mathbf{y}_{N_C}\}$, of N_C reference vectors in D^k. Usually, $D \equiv \Re$. Y is called *codebook* and its elements *codewords*. So, a vector quantizer can be represented as a function:

$$q : X \longrightarrow Y \tag{2.1}$$

Starting from this VQ definition, it is possible to obtain a partition $\mathcal{S}$ of the set X. It is constituted by the N_C subsets S_i of X:

$$S_i = \{\mathbf{x} \in X : q(\mathbf{x}) = \mathbf{y}_i\} \;\; i = \mathit{1}, \ldots, N_C \tag{2.2}$$

Generally, the subsets S_i are called “cells”.

2.2 Evaluation of the quantization error

The aim of the VQ is to represent the whole set X by the reduced one Y. When an element $\mathbf{x} \in X$ is approximated by a codeword $\mathbf{y}_i$, we have a quantization error because, generally, $\mathbf{x} \neq \mathbf{y}_i$.

The average QE, calculated for the whole data set, is a useful instrument to evaluate the effectiveness of a VQ. A VQ of X is better than another (always of X) when it has a lower average QE.

First of all, to establish how much $\mathbf{x}$ is different from $\mathbf{y}_i$ we need the definition of a distance (or distortion) operator d:

$$d : D^k \times D^k \longrightarrow \Re \tag{2.3}$$

The QE is the value assumed by $d(\mathbf{x}, \mathbf{y}_i)$, that is equivalent to $d(\mathbf{x}, q(\mathbf{x}))$.

Each quantizer is univocally determined after the codebook Y and the partition $\mathcal{S}$ have been fixed. Mean QE (MQE) $D(\{Y, \mathcal{S}\})$ (or $D(q)$), also called mean distortion, is defined [32]:

$$\begin{aligned} D(\{Y, \mathcal{S}\}) &\equiv D(q) = E\{d(\mathbf{x}, q(\mathbf{x}))\} = \\ &= \sum_{i=1}^{N_C} P(\mathbf{x} \in S_i)\ E\{d(\mathbf{x}, \mathbf{y}_i) \mid \mathbf{x} \in S_i\} \end{aligned} \tag{2.4}$$

where the symbols $P()$ and $E\{\}$ mean respectively probability and expectation value.

This definition has general validity. There exist two distinct alternatives.

2.2.1 Source whose statistical properties are known

If X is a continuous set we can use the integral operator. So, we have:

$$E\{d(\mathbf{x}, \mathbf{y}_i) \mid \mathbf{x} \in S_i\} = \int_{S_i} d(\mathbf{x}, \mathbf{y}_i) p(\mathbf{x} \mid \mathbf{x} \in S_i) d\mathbf{x} \tag{2.5}$$

$$P(\mathbf{x} \in S_i) = \int_{S_i} p(\mathbf{x}) d\mathbf{x} \tag{2.6}$$

where we indicate with the symbol $p()$ a probability distribution function.

Instead, if X is discrete we have to use the summation operator:

$$E\{d(\mathbf{x}, \mathbf{y}_i) \mid \mathbf{x} \in S_i\} = \sum_{n:\mathbf{x}_n \in S_i} d(\mathbf{x}_n, \mathbf{y}_i) P(\mathbf{x}_n \mid \mathbf{x} \in S_i) \tag{2.7}$$

$$P(\mathbf{x} \in S_i) = \sum_{n:\mathbf{x}_n \in S_i} P(\mathbf{x}_n) \tag{2.8}$$

2.2.2 Source whose statistical properties are unknown

We have a finite data set of N_P elements. Generally, they are called input vectors or learning patterns. If N_P is great enough and the elements are well distributed, i.e. they faithfully reproduce the source statistics, from eq. (2.4) derives:

$$E\{d(\mathbf{x}, \mathbf{y}_i) \mid \mathbf{x} \in S_i\} = \begin{cases} \frac{1}{N_i} \sum_{n:\mathbf{x}_n \in S_i} d(\mathbf{x}_n, \mathbf{y}_i) & \text{if } N_i \neq 0 \\ 0 & \text{if } N_i = 0 \end{cases} \tag{2.9}$$

$$P(\mathbf{x} \in S_i) = \frac{N_i}{N_P} \tag{2.10}$$

where N_i is the number of patterns belonging to the ith cell.

By employing 2.9 and 2.10, 2.4 can be expressed, in a more compact form, as follows:

$$\begin{aligned} D(\{Y, \mathcal{S}\}) &\equiv D(q) = E\{d(\mathbf{x}, q(\mathbf{x}))\} = \\ &= \frac{1}{N_P} \sum_{i=1}^{N_C} D_i \end{aligned} \tag{2.11}$$

where we indicate with D_i the ith cell total distortion:

$$D_i = \sum_{n:\mathbf{x}_n \in S_i} d(\mathbf{x}_n, \mathbf{y}_i) \tag{2.12}$$

2.3 Distortion measure

Various functions can be adopted as distance measures [32]. A very general definition that comprises the most frequently used quadratic measures, when $D \equiv \Re$, is:

$$d(\mathbf{x}, \mathbf{x}') = (\mathbf{x} - \mathbf{x}') \cdot \mathbf{W}(\mathbf{x}) \cdot (\mathbf{x} - \mathbf{x}')^t \tag{2.13}$$

where the squared matrix $\mathbf{W}(\mathbf{x})$ must be symmetrical and positive definite. From this definition, if $\mathbf{W}(\mathbf{x})$ is the k-dimensional identity matrix, the well known square Euclidean distance (or Squared Error, SE) follows:

$$d(\mathbf{x}, \mathbf{x}') = \sum_{i=1}^{k} (x_i - x_i')^2 \tag{2.14}$$

This measure is the most widely used in literature. While, if $\mathbf{W}(\mathbf{x})$ is a fixed diagonal matrix, the Weighted Square Error (WSE) follows:

$$d(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{k} w_i (x_i - y_i)^2 \tag{2.15}$$

where w_i (≥ 0) are the weights for each of the components.

2.3.1 Mean quantization error measures

The distortion measure permits the evaluation of the MQE. A very frequently adopted criterion is the Mean Squared Error (MSE). Its formulation, for a finite data set, is:

$$\text{MSE} = \frac{1}{N_P} \sum_{p=1}^{N_P} d(\mathbf{x}_p, q(\mathbf{x}_p)) \tag{2.16}$$

where $\mathbf{x}_p$ is the pth learning pattern and $d()$ is the SE. Sometimes, the square root of the MSE (RMSE) is used.

Other times, Normalized Mean Squared Error (NMSE) is adopted. It corresponds to the MSE divided by the MSE obtained with a codebook of only one codeword, $\mathbf{c}$, placed at the centroid[1] of the whole data set:

$$\text{NMSE} = \frac{\text{MSE}}{\frac{1}{N_P} \sum_{p=1}^{N_P} d(\mathbf{x}_p - \mathbf{c})} \tag{2.17}$$

[1]see section 2.4.2 for the definition of centroid

2.4 Optimal quantizer

A quantizer is optimum when, for each other quantizer with the same number of codewords, a higher MQE is found. In mathematical terms, q^* is optimum if, for each other q, we have $D(q^*) \leq D(q)$.

In the following, the author will describe the two main conditions which, from a mathematical point of view, are necessary so that a quantizer can be said to be optimum. The two conditions are usually called the Nearest Neighbour Condition (NNC) and the Centroid Condition (CC).

2.4.1 Nearest Neighbor Condition

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:

$$\begin{aligned} \bar{S}_i &= \{\mathbf{x} \in X : \; d(\mathbf{x}, \mathbf{y}_i) \leq d(\mathbf{x}, \mathbf{y}_j), \\ &\quad j = 1, ..., N_C, \; j \neq i\} \quad i = 1, ..., N_C \end{aligned} \tag{2.18}$$

The sets S_i just defined, constitute a partition of the input data set. This is the "Voronoi Partition" [35] and is refered to with the symbol $\mathcal{P}(Y) = \{\bar{S}_1, \cdots, \bar{S}_{N_C}\}$. As $\mathcal{P}(Y)$ must be a partition, when an input vector has the same distance from two or more codewords, it needs to choose a unique manner to assign this vector to only one S_i.

NNC permits us to obtain an optimal partition [32], i.e. for every partition $\mathcal{S}$ of the input data set, it holds:

$$D(\{Y, \mathcal{S}\}) \geq D(\{Y, \mathcal{P}(Y)\}) \tag{2.19}$$

2.4.2 Centroid Condition

Given a fixed partition $\mathcal{S}$, the CC concerns the procedure to find the optimal codebook.

Let us define centroid or center of gravity of a given set $A \subseteq D^k$ the vector $\bar{\mathbf{x}}(A)$ for which:

$$E\{d(\mathbf{x}, \bar{\mathbf{x}}(A)) \mid \mathbf{x} \in A\} = \min_{\mathbf{u} \in D^k} E\{d(\mathbf{x}, \mathbf{u}) \mid \mathbf{x} \in A\} \tag{2.20}$$

For example, if $A \subset \Re^2$ and d is the squared Euclidean distance, then $\bar{\mathbf{x}}(A)$ coincides with the geometrical center of gravity of the set A. More in general, when $D \equiv \Re$, the number of elements of A is N_A and the squared euclidean distance is adopted (2.14), we have:

$$\bar{\mathbf{x}}(A) = \frac{1}{N_A} \sum_{\mathbf{x} \in A} \mathbf{x} \tag{2.21}$$

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

$$\bar{X}(\mathcal{S}) \equiv \{\bar{\mathbf{x}}(S_i); i = i, ..., N_C\} \tag{2.22}$$

it is optimum [32], i.e. for every codebook Y, it holds:

$$D\{Y, \mathcal{S}\} \geq D(\{\bar{X}(\mathcal{S}), \mathcal{S}\}) \tag{2.23}$$

Chapter 3

Basic algorithms for hard and fuzzy CA/VQ

In this chapter two basic techniques for CA/VQ will be presented: the Generalized Lloyd Algorithm and the Fuzzy c-Means Algorithm (FCM). The former belongs to the hard family, the latter to the fuzzy one. Besides, a brief comparison between the hard and the fuzzy approach is given [2].

3.1 Generalized Lloyd Algorithm (GLA) or LBG

In 1980 Linde, Buzo and Gray [32] proposed an improvement of the Lloyd's technique [81]. They extended Lloyd's results from mono- to k-dimensional cases. For this reason their algorithm is known as the Generalized Lloyd Algorithm (GLA) or LBG from the initials of its authors.

In a few words, the LBG algorithm is a finite sequence of steps in which, at every step, a new quantizer, with a total distortion less or equal to the previous one, is produced.

Now, we will describe the LBG steps. We can distinguish two phases, as shown in Fig. 3.1: the initialization of the codebook and its optimization. In the initialization phase two methods are mainly used: random and by splitting.

Firstly, we will describe the optimization step. It will simplify the LBG explanation. In fact, several concepts necessary to describe this step are useful for the initialization phase, too. In the following we will use these symbols:

- m: iteration number;

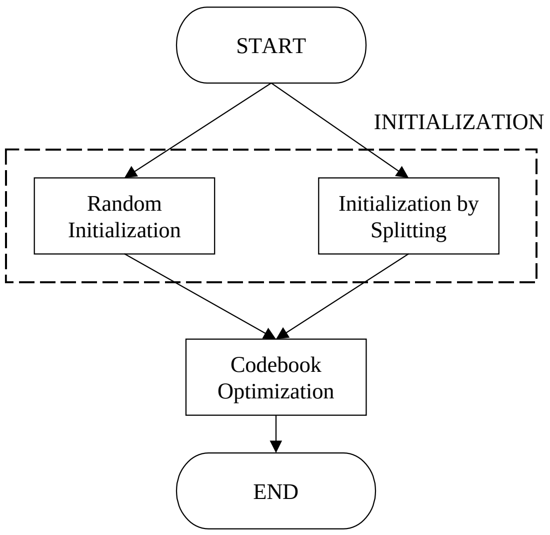


Figure 3.1: The LBG procedure

- Y_m: mth codebook;
- D_m: MQE calculated at the end of the mth iteration.

3.1.1 Codebook optimization

Fig. 3.2 shows the high-level flow-chart. The codebook optimization starts from an initial codebook and, after some iterations, generates a final codebook with a distortion corresponding to a local minimum.

1. **Initialization.** The following values are fixed:

   - N_C: number of codewords;
   - $\epsilon \geq 0$: precision of the optimization process;
   - Y_0: initial codebook;
   - $X = \{\mathbf{x}_j;\ j = \mathit{1}, ..., N_P\}$: learning patterns;

   Further, the following assignments are made:

   - $m = 0$;
   - $D_{-1} = +\infty$;

2. **Partition calculation.** Given the codebook Y_m, the partition $\mathcal{P}(Y_m)$ is calculated according to the NNC (2.18).

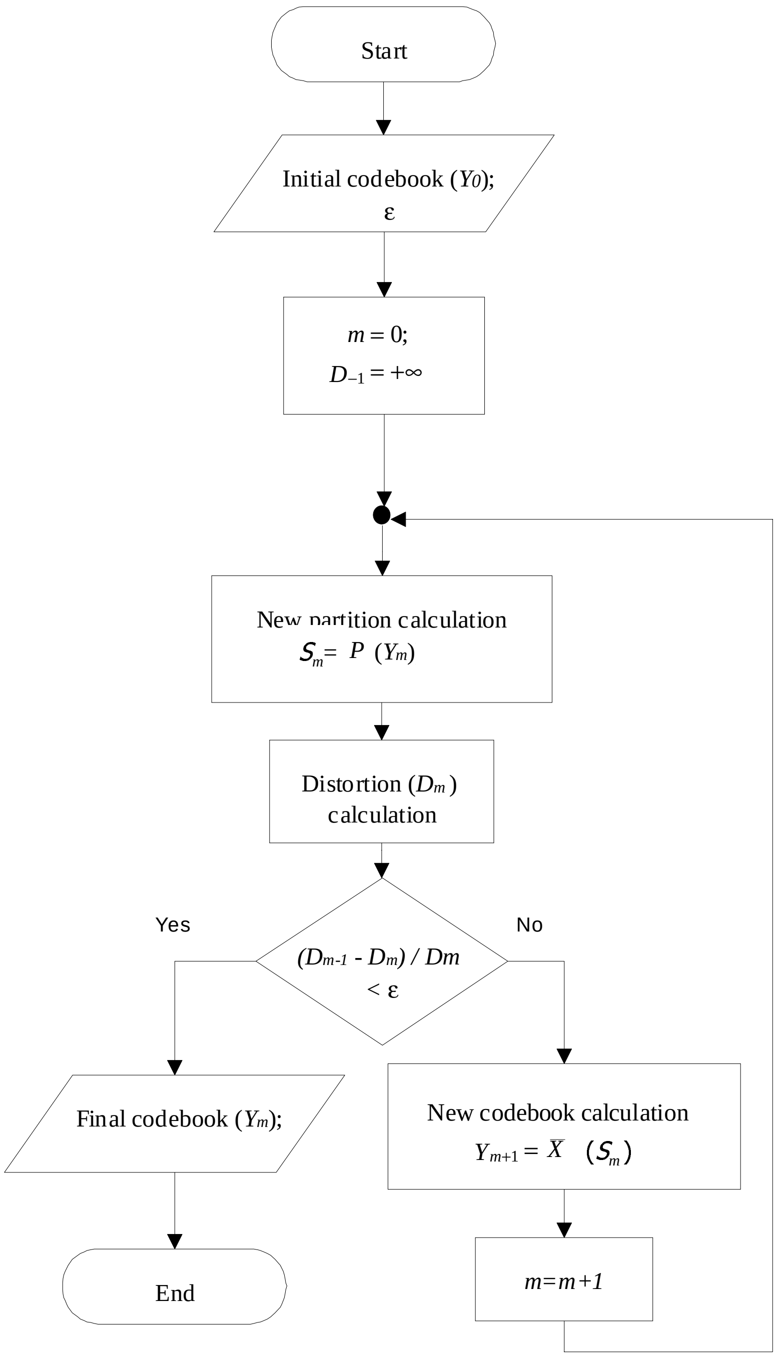

Start
Initial codebook (Y_0); ε
$m = 0$; $D_{-1} = +\infty$
New partition calculation $S_m = P(Y_m)$
Distortion (D_m) calculation
$(D_{m-1} - D_m) / Dm < \varepsilon$
Yes
No
Final codebook (Y_m);
End
New codebook calculation $Y_{m+1} = \bar{X}(S_m)$
$m=m+1$

Figure 3.2: LBG codebook optimization

3. **Termination condition check.** The quantizer distortion ($D_m = D(\{Y_m, \mathcal{P}(Y_m)\})$ is calculated according to eq. (2.4). If $\mid (D_{m-1} - D_m) \mid / D_m \leq \epsilon$ then the optimization ends and Y_m is the final returned codebook[1].

4. **New codebook calculation.** Given the partition $\mathcal{P}(Y_m)$, the new codebook is calculated according to the CC (2.22). In symbols:

$$Y_{m+1} = \bar{X}(\mathcal{P}(Y_m)) \tag{3.1}$$

   After, the counter m is increased by one and the procedure follows from step 2.

In [32] it is demonstrated that these steps assure that the series D_m is not increasing and convergent.

3.1.2 Initialization of the codebook

The codebook initializiation is a very important task. In fact, a bad choice of the initial codewords generally leads to a final quantizer with a high MQE. Here, we describe the random initialization and the initialization by splitting.

- **Random initialization**. The initial codewords are randomly chosen [55]. Generally they are chosen inside the convex hull of the input data set.

- **Initialization by splitting**. This initialization requires that the number of codewords is a power of 2. The procedure starts from only one codeword that, recursively, splits it in two distinct codewords [32]. More precisely, the generic mth step consists in the splitting of all vectors obtained at the end of the previous step. After the splitting, an optimization step is executed according to the method described in the sub-section 3.1.1.

  The splitting criterion is shown in Fig. 3.3. It starts from one codeword $\mathbf{y}$. It splits this vector into two close vectors $\mathbf{y} + \mathbf{e}$ and $\mathbf{y} - \mathbf{e}$ where $\mathbf{e}$ is a fixed perturbation vector.

[1]The termination condition depends both on the ϵ value and the adopted distortion measure. It is meaningless to specify only the ϵ value because, with two different distortion measures (as, for example, the MSE and the RMSE), the expected value of the number of iterations can drastically change and, of course, the final mean distortion value. In this thesis, each time we specify an ϵ value, it refers to the RMSE. A typical range of values for ϵ is $[0.001, 0.1]$.

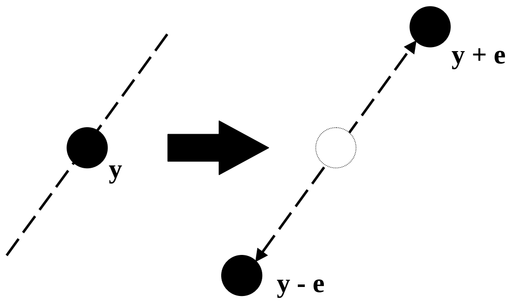


Figure 3.3: Splitting of a codeword

These techniques are not the only ones present in literature. From the others, we cite the maximum distance initialization [82].

3.2 Considerations about the LBG algorithm

The algorithm just presented usually finds a locally optimum quantizer. The main problem is that, often, this optimum is very far from an acceptable solution.

If we qualitatively comment the analytical expressions regarding the codeword adjustment we could say that at each iteration codewords "move" through contiguous regions. This implies that a bad initialization could lead to the impossibility of finding a good quantizer.

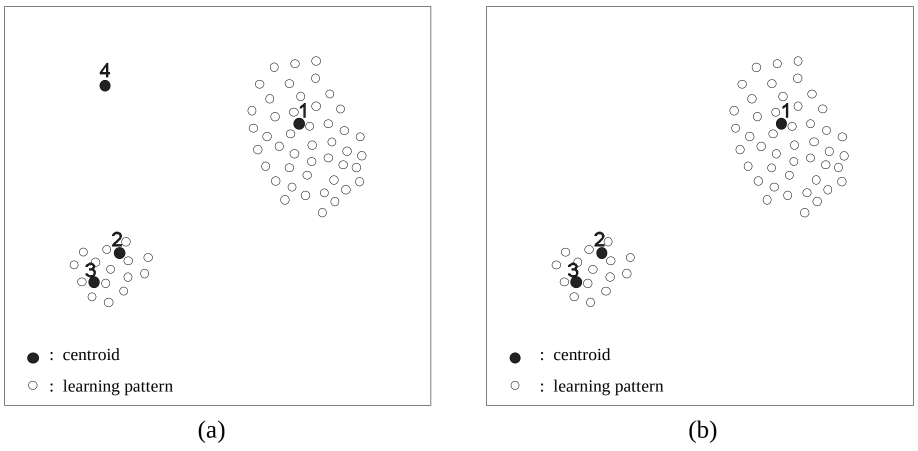


Figure 3.4: Badly positioned centroids

For example, let us examine Fig. 3.4. On the left side, part (a), we see the codeword number 4. According to the (2.18), it will always generate an empty cell because all the elements of the data set are nearer to the other codewords. So, following the steps of the traditional LBG, it cannot move and will never represent any element. For this reason we can say it is useless. The same authors of the LBG [32] proposed some solutions to this problem such as the assigning of the codeword to a non-empty cell.

But we think there is another problem that strongly limits the classical LBG and its solution appears a difficult task. Let us look at the right side, part (b), of Fig. 3.4. This configuration shows two clusters and three codewords. In the little cluster there are two codewords whereas, in the other, only one. The elements in the data set in the smaller cluster are all well approximated by the two related codewords. Instead, a lot of elements in the larger one are badly approximated by the related codeword. For this geometrical distribuition, it would be preferable that two codewords were inside the big cluster and only one in the other, but the LBG optimization algorithm, in this situation, does not permit the migration of a codeword from the little cluster to the big one. This is a great limitation.

To improve the performance of the LBG algorithm, we think that it is crucial to develop a criterion that identifies these situations. Further, it must be able to find which codewords it is better to move and where they have to be placed, without any contiguity limitation.

Some authors already introduced some interesting criterions [38].

3.3 Fuzzy techniques

The main difference between hard and fuzzy techniques is that, during the construction of the codebook, each vector can belong to several clusters with different degrees of membership or not. They are a generalization of hard ones.

In fuzzy techniques, the objective is the minimization of a functional similar to (2.4), but containing the membership degrees, too. For example, in fuzzy c-Means (FCM) [49], it is:

$$\min_{(U,Y)} \left\{ J_m(U,Y,X) = \sum_{h=1}^{N_P} \sum_{i=1}^{N_C} (u_{ih})^m \|\mathbf{x}_h - \mathbf{y}_i\|_A^2 \right\} \tag{3.2}$$

where:

- $U = \{u_{ih}\}$ is the matrix with the fuzzy labels. Its generic element u_{ih}

is the degree of membership of the h-th pattern to the i-th class. The matrix U must be arrayed in such a way that each column of U is a fuzzy label, i.e.:

$$u_{ih} \in [0,1] \ \ \forall i, h \qquad \textstyle\sum_{i=1}^{N_C} u_{ih} = 1 \ \ \forall h$$

- $Y = \{\mathbf{y}_{\mathit{1}}, \mathbf{y}_{\mathit{2}}, ..., \mathbf{y}_{N_C} \mid \mathbf{y}_i \in \Re^p \ i = \mathit{1}, \mathit{2}, ..., N_C\}$ is the codebook;
- $||\mathbf{x}||_A = \sqrt{\mathbf{x}^T A \mathbf{x}}$; A is a positive definite $p \times p$ matrix. If A is the identical matrix ($A = I$) the expression $||\mathbf{x}||_A$ is the Euclidean distance.

The functional (3.2) is minimized almost in the same manner used in the LBG. A complete description can be found in [49, 50]. The main problem to solve in FCM is the right choice of m. In [49] the interval $[1.1, 5]$ is suggested.

In [83] the algorithm is analyzed for $m \to 1$ and $m \to \infty$. While complete formulas can be found in [49], here we remind that, for $m \to 1$, the algorithm becomes a hard K-means one. While, for $m \to \infty$, all of the codewords move towards the grand mean (i.e. the global mean value) of the input vectors.

The objective function of FLVQ [49] is similar to (3.2) and, also in this case, the result is dependent on the choice of m. If we consider FALVQ [53, 54], the parameters to choose are more than one and, according to their choices, the algorithm could also not be convergent.

As shown in [49, 53], when the configuration parameters are well chosen, these techniques are less sensitive to initial conditions than hard ones.

3.4 Considerations about hard and fuzzy techniques

As regards the dependence from the initial conditions, the behavior of fuzzy techniques in case of badly positioned codewords (see Fig. 3.4) is different than hard ones. In fact, each pattern attracts all of the codewords and not only the nearest of all. The degree of attraction is greater for the nearest codewords and lower for the farthest ones. In this way, all of the codewords can be attracted by all of the patterns thus allowing migrations that are not allowed in hard techniques. However, we must remember that the performances depend strongly on the choice of the configuration parameters. For example, in [50] it is shown that both the results and the number of iterations required to obtain a good result change when m varies.

We have also to consider that in several applications, such as classification [49, 50, 53, 54] or image compression [53], the codebook is obtained by fuzzy algorithms but the input patterns are classified or approximated according to the nearest neighbor condition (section 2.4.1), i.e. in a crisp manner, where each pattern belongs to one and only one cell.

Chapter 4

The Enhanced LBG Algorithm

4.1 Introduction

In this chapter we present the new algorithm we called Enhanced LBG (ELBG) [1, 3]. It belongs to the hard and K-means VQ groups and derives directly from the simpler LBG. The basic idea we developed is the concept of utility of a codeword. Even if some authors already introduced the utility [38], our definition, meaning and computational complexity are totally different. The utility index we use is a powerful instrument to overcome some of the greatest drawbacks of the LBG and other VQ algorithms. As we have already stated, one of the main problems is that, in the case of a bad choice of the initial codebook, generally, the results are not good. The utility allows a good identification of these situations. Further, it permits the recognition of the badly positioned codewords and gives us useful indications about regions where they should be placed. This work, like [59], has been inspired by Gersho's theorem [84]. This theorem states that, if some hypothesis are verified, the distortion associated to each codeword is the same as the others in an optimal codebook. In the same way, ELBG looks for a codebook to which each codeword contributes in the same manner, i.e. the utility of all the codewords is the same. Basically, our algorithm is an LBG in which some further steps have been added. Starting from some mathematical properties of the utility index and other considerations, we developed some sub-optimal operations on which ELBG is based. We adopted these sub-optimal operations in order to not increase the final overhead of the ELBG too much.

The experimental results we have reached show that ELBG is able to find better codebooks than previous works and the computational complexity is virtually the same as the simpler LBG algorithm.

The chapter is organized as follows:

- in Sections 4.2-4.6 the algorithm is described;
- in Section 4.7 results and comparisons with other algorithms are presented;
- lastly, Section 4.8 contains the author's conclusions.

4.2 General considerations

The algorithm we propose is an attempt to find a solution for the two drawbacks of the classical LBG we discussed in section 3.2. We will formally introduce a new quantity that we call the utility of a codeword. It allows us to deal with both drawbacks described in the previous section from a unique point of view. As we will explain in the next sections, the terms utility of a codeword and utility of a cell have the same meaning. In the following, we will suppose that we are dealing with a finite input data set $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_{N_P}\}$.

Fig. 4.1 shows the high-level flow-chart of the ELBG algorithm. The only difference between LBG and ELBG is the ELBG block. The functionalities of the ELBG block are summarized in Fig 4.2. First of all, there is the utility evaluation. After the evaluation of the utilities of the codewords, we identify the cells with a low utility. This information is very useful for the next step: the smart shifting of codewords. We try to shift all the low-utility codewords near to the ones with high utility. Each attempt leading to a lower MQE is confirmed. The aim of these operations is to obtain the equalization of the total distorsions related to cells (D_i, see eq. (2.12)), as one of Gersho's theorems suggests [33, 84]. As we will see in detail in the next sections, this allows us to overcome the drawbacks we exposed in section 3.2.

Here, we only say that the heart of the ELBG block is the execution of several Shifting of Codewords Attempts (SoCA's). When a SoCA produces a decrease in the MQE, then the SoCA is confirmed. In this case we say that a Shift of Codeword (SoC) is executed. If we do not have any MQE decrease, the shift is discarded.

Besides, we wish to underline that all the additional steps we introduced in the LBG algorithm to obtain the ELBG are very efficient. The term "efficient" refers to a low computational complexity operation. So, the overhead we introduced in the original LBG is negligible, as will be shown in section 4.6.

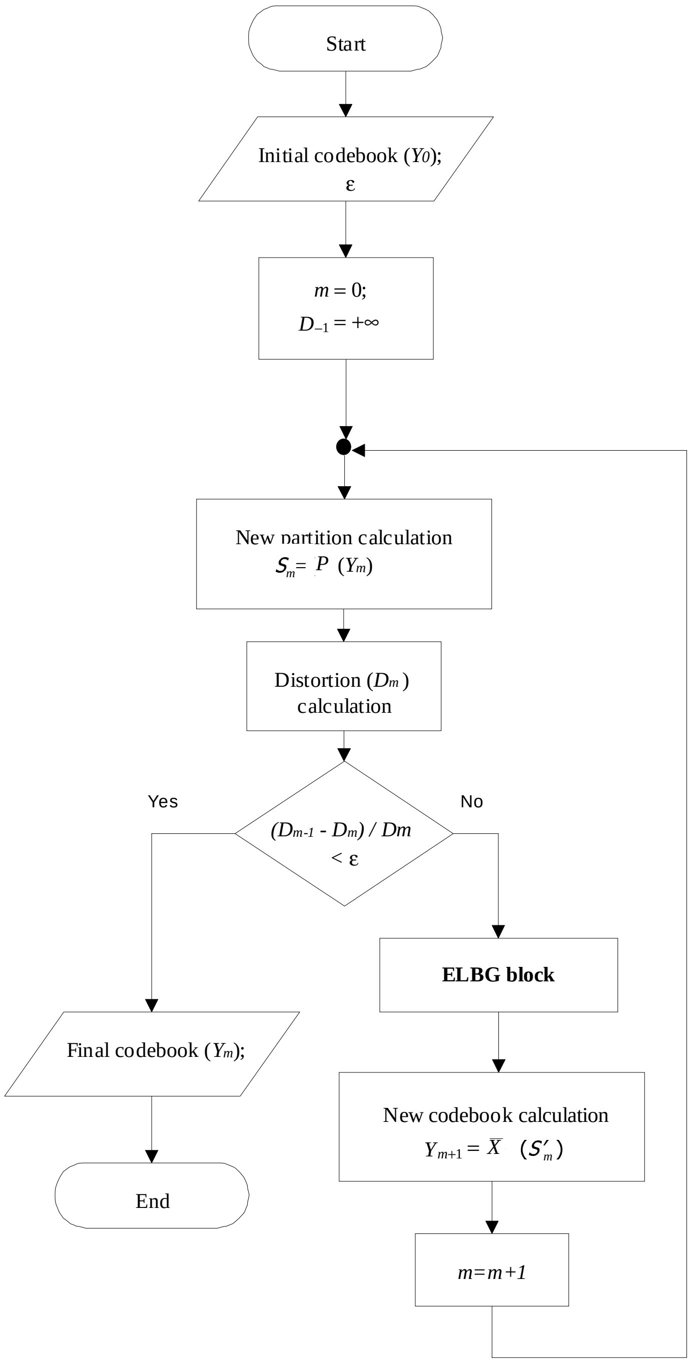

Start
Initial codebook (Y_0); ε
$m = 0$; $D_{-1} = +\infty$
New partition calculation $S_m = P(Y_m)$
Distortion (D_m) calculation
$(D_{m-1} - D_m) / D_m < \varepsilon$
Yes
No
ELBG block
Final codebook (Y_m);
New codebook calculation $Y_{m+1} = \bar{X}(S'_m)$
End
$m=m+1$

Figure 4.1: ELBG codebook optimization

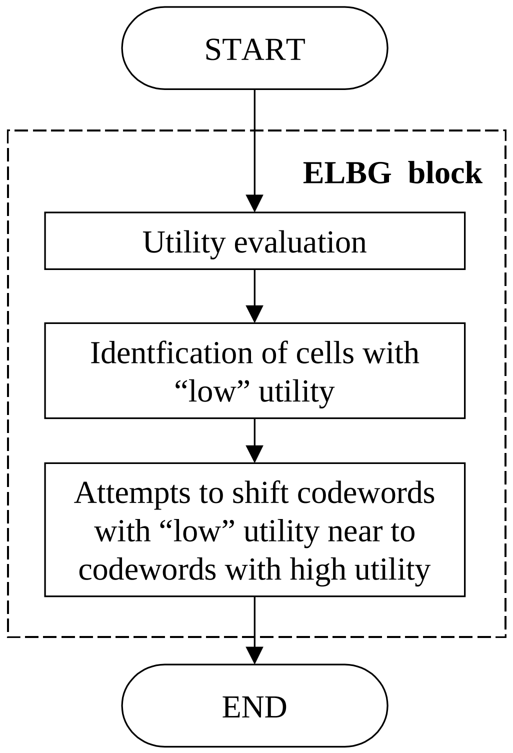

START
ELBG block
Utility evaluation
Identfication of cells with "low" utility
Attempts to shift codewords with "low" utility near to codewords with high utility
END

Figure 4.2: High-level flow-chart of the ELBG block

4.3 Distortion equalization and Utility

The idea of the utility was suggested to us by one of Gersho's theorems [33] where he explained his partial distortion theorem [84] 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 [84], a high resolution quantizer has a number of codewords tending to infinite). But, in [59], 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" (U_i) of the ith cell as the value of D_i normalized with respect to its mean value (D_{mean}). In formal terms, we have:

$$D_{\text{mean}} = \frac{1}{N_C} \sum_{i=1}^{N_C} D_i \tag{4.1}$$

$$U_i = \frac{D_i}{D_{\text{mean}}} \quad i = 1, ..., N_C \tag{4.2}$$

In the following, we will use both the term utility index of a cell and utility index of a codeword. Substantially, there is no difference between the two terms. In fact, we can use equation (4.2) only if a cell is considered together with the related codeword and vice versa. We will often use only the shorter term "utility".

According to the definition just given, the equalization of the distorsions is equivalent to the equalization of the utilities.

Our idea is to obtain the desired equalization by joining, for each SoCA, a low-utility (lower than 1) cell with a cell adjacent to it, hoping to obtain a bigger cell whose utility is closer to 1 than before. At the same time, we split a high-utility (higher than 1) cell into two smaller ones whose utilities, are, if possible, closer to 1 than the big cell. We can say that this operation is equivalent to move the low-utility codeword inside the high-utility cell. If we refer to Fig. 3.4. (a) we see that the utility of cell 4 (U_4) is 0, that U_2 and U_3 are lower than 1 and that U_1 is greater then 1. These values, according to the possibility illustrated above, suggest moving the 4th codeword near the 1st codeword. Instead, if we see Fig. 3.4. (b) we should move codeword 2 or 3 near the 1st one. This is a "smart" manner of shifting codewords that allows their migration through non-contiguous regions.

Fig. 4.8 shows a typical distribution of the utility indexes when an initial random codebook is given. It is a very widespread "bell". The shifting of the codewords on the left side of the figure near to the ones on the right side produces an adjustment of the original bell into a narrow one, as is shown in Fig. 4.10. However, we must remember that our primary objective is the MQE minimization. So, we execute a SoC only when we are sure that it produces a mean distortion decrease. The way we execute a SoC and the evaluation of its effect on the QE are the argument of the next subsections.

The considerations exposed in this sub-section allowed us to develop an objective criterion to select the codewords to be shifted and the cells where they have to be placed, as will be explained in the next sub-sections.

4.4 Detailed description of the ELBG block

Fig. 4.3 details the previous Fig. 4.2 in which the main steps of the ELBG block are illustrated.

The ELBG consists in the execution of a certain number of SoCA's. In what follows, we will describe in detail a SoCA. We will use Figs. 4.4, 4.5, 4.6, 4.7. They represent a SoCA for a simple bi-dimensional problem.

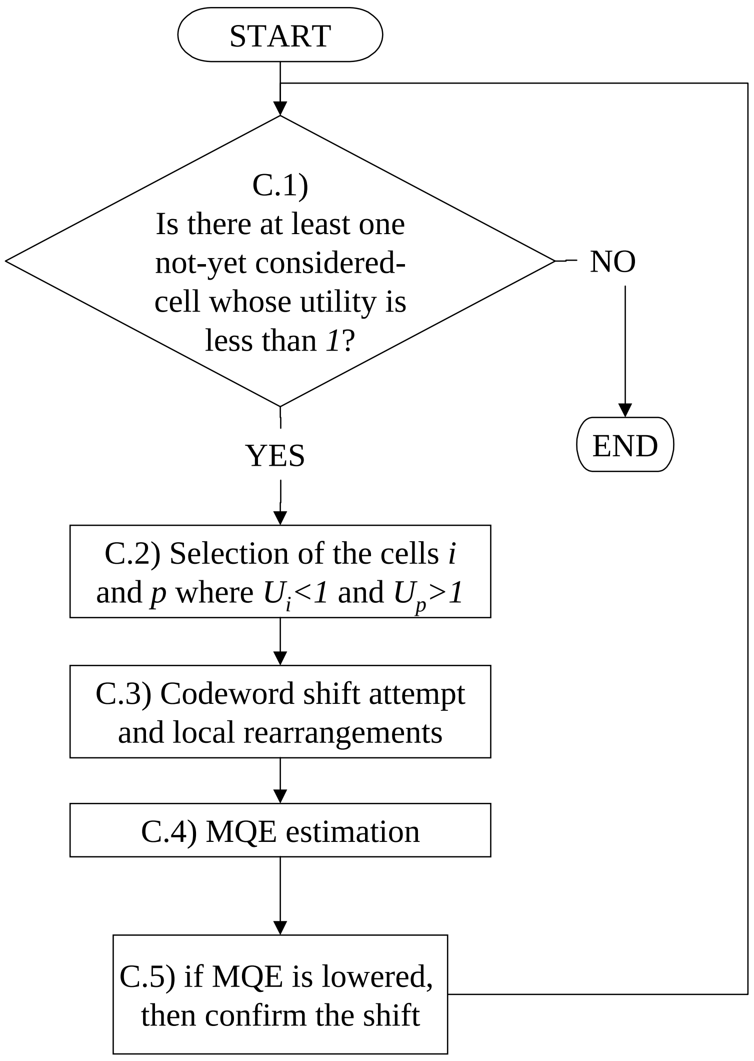

START
C.1) Is there at least one not-yet considered-cell whose utility is less than *1*?
NO
END
YES
C.2) Selection of the cells *i* and *p* where $U_i<1$ and $U_p>1$
C.3) Codeword shift attempt and local rearrangements
C.4) MQE estimation
C.5) if MQE is lowered, then confirm the shift

Figure 4.3: Detailed description of the ELBG block

4.4.1 Termination condition

The first condition in the upper part of Fig. 4.3 regards the termination of the whole ELBG block. We check to see if at least one cell has a utility index lower than 1 and it has not been involved in previous shifts. If no cell exists, then the algorithm ends. Otherwise the next steps regarding a new SoCA follow.

4.4.2 Selection of cells

This step is necessary to recognize all the cells involved in the current SoCA. We look for two different cells.

- One cell must have a utility index less than 1. We will refer to it as the ith cell.
- One cell must have a utility index greater than 1. We will indicate it as the pth cell.

The ith cell, S_i, is searched for in a sequential manner. Instead, the pth cell is looked for in a stochastic way. The method adopted sounds like the roulette wheel selection in genetic algorithms [85]. Practically, we choose a cell with a probability P_p proportional to its utility value. In mathematical terms:

$$P_p = \frac{U_p}{\sum_{h:U_h>1} U_h} \tag{4.3}$$

4.4.3 Codeword shift and local rearrangements

This step consists in a SoCA. We try to shift the codeword $\mathbf{y}_i$ near $\mathbf{y}_p$, i.e. the codewords related to the cells S_i and S_p respectively. This situation is illustrated in Fig. 4.4 for our bi-dimensional problem.

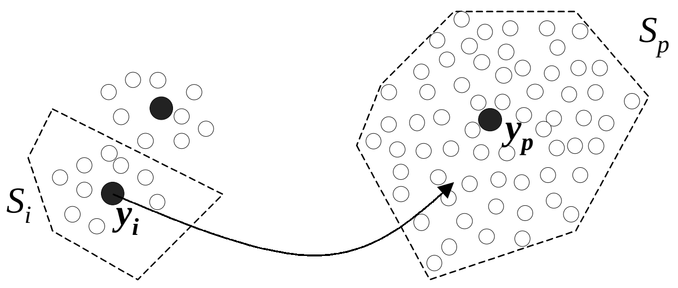


Figure 4.4: Initial situation before the SoCA

A similar shift produces a new codebook. In the traditional LBG, after a new codebook generation, the calculation of the partition satisfying the NNC, i.e. the Voronoi partition, follows. As we have several SoCA's and each of them must be evaluated, we would have to introduce a very heavy overhead into the classical LBG[1]. Our aim is to improve the LBG with a low overhead, so we avoid recalculating the Voronoi partition.

To simplify the overall procedure and, of course, to drastically reduce the overhead, we suppose that, after the shift, in the new partition only the patterns related to S_i and S_p will be subject to change. In our algorithm, these patterns will be the only ones with related codewords different from before. We shift $\mathbf{y}_i$ near $\mathbf{y}_p$. As $\mathbf{y}_p$ is, generally, localized at the center of S_p, we think that it is better to move $\mathbf{y}_p$, too. So, it is possible to distribute the two codewords inside the S_p cell in a better way. The solution we have found is very simple. It does not assure a better distribution, but we made a lot of experimental trials that have shown its validity.

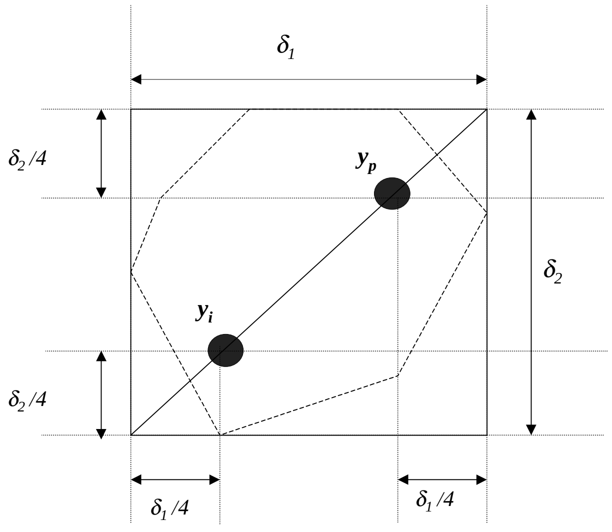


Figure 4.5: The cell S_p and the hyperbox containing it

As a finite number of k-dimensional vectors forms the input data set, the generic pth cell is contained in the k-dimensional hyperbox I_p:

$$I_p = [x_{1\mathrm{m}}, x_{1\mathrm{M}}] \times [x_{2\mathrm{m}}, x_{2\mathrm{M}}] \times ... \times [x_{k\mathrm{m}}, x_{k\mathrm{M}}] \tag{4.4}$$

[1]We must remember that the most onerous step in the LBG algorithm is precisely the Voronoi partition calculation.

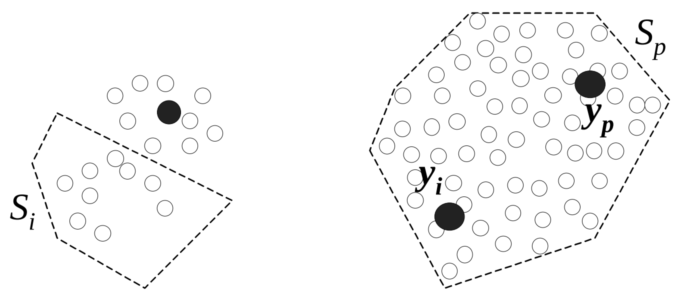


Figure 4.6: Codewords position immediately after the shift

where $x_{h\mathrm{m}}$ and $x_{h\mathrm{M}}$ are respectively the minimum and maximum value assumed by the hth dimension of all patterns belonging to S_p. We place $\mathbf{y}_i$ and $\mathbf{y}_p$ on the principal diagonal of I_p. We divide the diagonal in three parts. Two are equal to the half of the central one. We place the two codewords at the ends of the central part as is illustrated in Figs. 4.5 and 4.6.

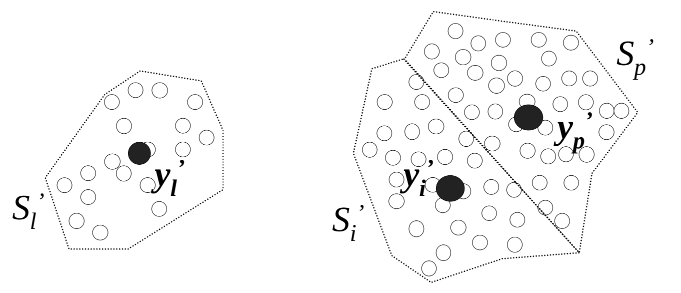


Figure 4.7: Codewords position and patterns distribution after the local rearrangements

Afterwards, $\mathbf{y}_i$ and $\mathbf{y}_p$ are adjusted with a local traditional LBG with a high value for ϵ (typically $0.1 \div 0.3$), so only a very few iterations (one or two) are generally executed. The codebook to be optimized contains only $\mathbf{y}_i$ and $\mathbf{y}_p$ and the input data set is S_p. The result of this optimization step is two new codewords ($\mathbf{y}_i'$ and $\mathbf{y}_p'$) and two new cells (S_i' and S_p'). The new codewords and the new cells substitute the corresponding ones in the old codebook and partition respectively. We show this in the right part of Fig. 4.7.

After the patterns belonging to S_p have been rearranged, we still have to rearrange the patterns of S_i because the related codeword has been moved away. We must again remember that the optimum way to assign these vectors is to calculate the Voronoi partition of the whole data set. As we want to avoid this operation, we adopt another sub-optimal low-complexity operation.

As shown in Fig. 4.7, we assign all vectors in S_i to S_l, $\mathbf{y}_l$ being the nearest codeword to $\mathbf{y}_i$. Afterwards, $\mathbf{y}_l$ is substituted by the centroid $\mathbf{y}'_l$ of the new set. In symbols:

$$\begin{cases} S'_l = S_l \cup S_i; \\ \mathbf{y}'_l = \bar{\mathbf{x}}(S'_l) \end{cases} \tag{4.5}$$

Generally, this solution is sub-optimal because in the Voronoi partition, the S_i vectors could distribute themselves among more cells.

4.4.4 Mean Quantization Error estimation

Now we have to understand if the SoCA produces a lowering of the MQE. If it does, then it is confirmed, i.e. it turns into a SoC. Otherwise the SoCA is rejected.

For an exact evaluation of the MQE, the calculation of the Voronoi partition is necessary. But, as we have already stated, this calculation will introduce a very high overhead. For this reason we employ a sub-optimal, but efficient solution again. It consists in the overestimation of the MQE we would obtain by finding the Voronoi partition. If the overestimated MQE is lower than the previous MQE (i.e the MQE we had before the SoCA) then we are sure that the shift produces an actual decrease in the final MQE. None of these operations require the calculation of the Voronoi partition. Thanks to this trick, the overhead introduced by the ELBG block is negligible in comparison to the time required by the standard LBG.

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, we can understand if the SoCA must be confirmed.

Let us remember that Y and $\mathcal{S}$ are the codebook and the partition before the shift. Y' and $\mathcal{S}'$ are the codebook and the partition we have after the shift. Y' is obtained by replacing in Y the three codewords $\mathbf{y}_i$, $\mathbf{y}_p$, and $\mathbf{y}_l$ with $\mathbf{y}'_i$, $\mathbf{y}'_p$, and $\mathbf{y}'_l$ respectively. In the same way, by substituting the related cells in $\mathcal{S}$, we obtain $\mathcal{S}'$.

The following symbols will be used:

- D_{old} is the MQE before the shift:

$$D_{old} = D(\{Y, \mathcal{S}\}) \tag{4.6}$$

- D_{new} is the MQE we would have by considering Y' and the Voronoi partition deriving from it :

$$D_{new} = D(\{Y', \mathcal{P}(Y')\}) \tag{4.7}$$

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

$$d_{old} = D_i + D_l + D_p \tag{4.8}$$

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

$$d_{new} = D'_i + D'_l + D'_p \tag{4.9}$$

If we calculated the Voronoi partition deriving from the new codebook Y', we would understand if the SoCA is useful or not, i.e. if $D_{new} \leq D_{old}$ or not.

Now, we will prove that, if we calculate only d_{new} and d_{old} and $d_{new} \leq d_{old}$, then we are sure that $D_{new} \leq D_{old}$. Our condition is only sufficient. For this reason we say that our algorithm is sub-optimal.

Let us suppose we have performed a SoCA as explained in the current and in the previous sections. Let us suppose that all the hypothesis made in these sections are true. We want to demonstrate that:

$$\text{if } d_{new} \leq d_{old}, \text{ then } D_{new} \leq D_{old}$$

Let us indicate $d_{const} = (N_P D_{old} - d_{old})$. The quantity d_{const} is the total distortion derived from the whole codebook and codewords eliminating the codewords and the patterns related to the ith, pth and lth cells. It remains constant from the old codebook to the new one. So:

$$D(\{Y', \mathcal{S}'\}) = \frac{1}{N_P}(d_{new} + d_{const}) \leq \frac{1}{N_P}(d_{old} + d_{const}) = D_{old}.$$

But, from the NNC, we know that:

$$D(\{Y', \mathcal{P}(Y')\}) <= D(\{Y', \mathcal{S}'\}) \ \ \forall \mathcal{S}'.$$

So we obtain:

$$D_{new} = D(\{Y', \mathcal{P}(Y')\}) \leq D_{old}$$

as we wished to demonstrate.

4.4.5 Confirmation or discarding of the SoCA

If $d_{new} \leq d_{old}$ the SoCA is confirmed. Therefore, we have a SoC. Otherwise the attempt is discarded. After, we try to effect another SoCA, i.e. we go back to point 4.4.1.

We can execute any number of consecutive SoC's when there is a decrease in the mean distortion. The final codebook and the related Voronoi partition introduce a mean distortion that is less than that we have obtained before any shift.

This approximation is, in the author's opinion, a good compromise between computational effort and precision. The value of the idea is confirmed by the experiments that we will illustrate in the next sections.

4.5 Considerations on the utility concept

In this Section we will discuss the utility concept. Starting from a case study we will examine the utility behaviour in the LBG case and in the ELBG.

We took the well-known Lena's image [86] of 512×512 pixels of 256 grey levels. The image was divided into 4×4 blocks and the resulting 16384 16-dimensional vectors were used as a data set. We fixed $N_C = 256$.

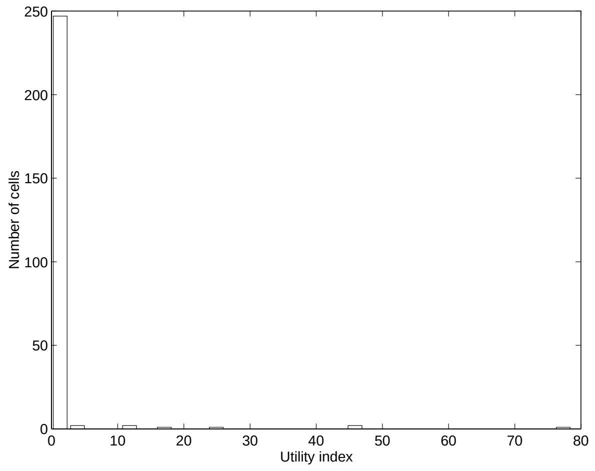


Figure 4.8: The initial distribution of the utility indexes

We generated a random initial codebook. In Fig. 4.8 the very wide and strongly non-symmetrical initial distribution of the utility indexes is shown. The related RMSE is 201.

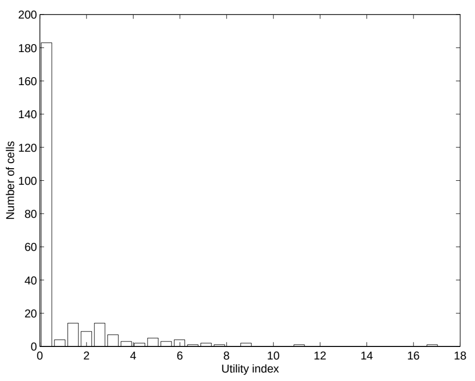


Figure 4.9: The final distribution of the utility indexes when the LBG algorithm is used

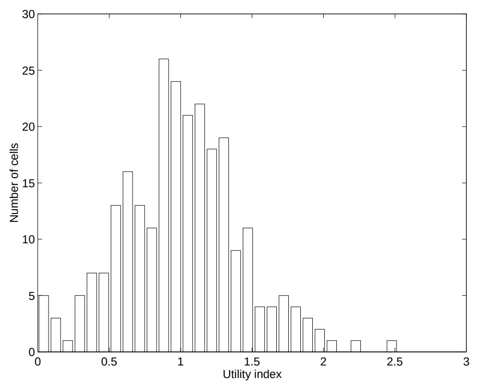


Figure 4.10: The final distribution of the utility indexes when the ELBG algorithm is used

Successively, we used the standard LBG algorithm. It required 18 iterations. The final RMSE was 33.4. The ELBG with the same initial codebook required only 11 iterations and the final RMSE was 25.8. In both algorithms we fixed $\epsilon = 0.001$.

In Figs. 4.9 and 4.10 the final distributions of the utility indexes are shown. In the LBG case we find a lot of codewords with a utility of almost 0 and some with very high utility values, up to $16 \div 18$. In the other case we have a more compact distribution. All utility values are comprised in the real interval $[0, 3]$.

In the following, we will not use the concept of standard deviation because we are dealing with several non symmetrical distributions. We prefer to use the concepts of left and right standard deviations (σ_l, σ_r) [87]. They are obtained calculating the standard deviation only of the values below and above the mean value respectively.

Figs. 4.11 and 4.12 show both σ_l and σ_r versus the number of iterations in the LBG case. σ_l is almost constant to 1 for all the iterations whereas σ_r decreases up to 3.72. It reaches about 90% of its total decrease in 4 iterations and slowly continue to decreases up to the final iteration.

Figs. 4.13 and 4.14 show both the standard deviations versus the number of iterations in the ELBG case. σ_l decreases in only 3 iterations to almost its final value of about 0.41. σ_r decreases up to 0.43 in only 3 iterations. So, ELBG effectively succeeds in shifting codewords with very low utility indexes near to codewords with very high utility indexes. Further, it "equalizes" the utility distribution. In fact, we have the total deviation equal to 0.42, i.e. $\sigma \simeq \sigma_l \simeq \sigma_r$.

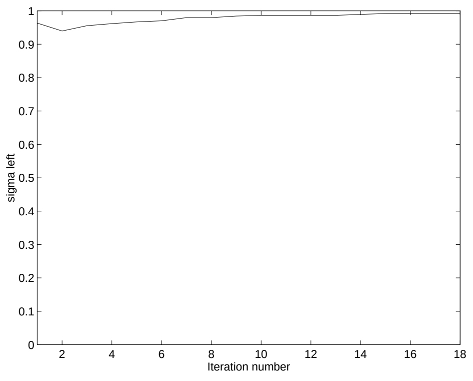


Figure 4.11: σ_l versus the number of iterations in the LBG case

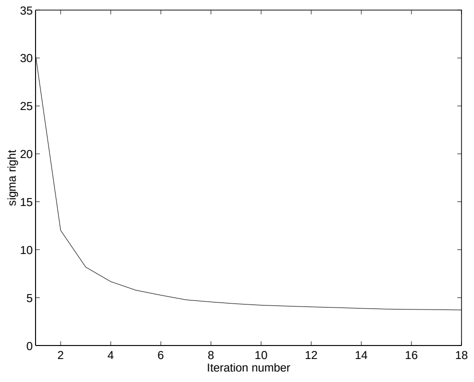


Figure 4.12: σ_r verus the number of iterations in the LBG case

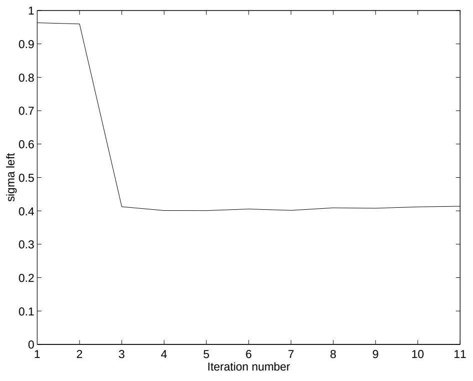


Figure 4.13: σ_l verus the number of iterations in the ELBG case

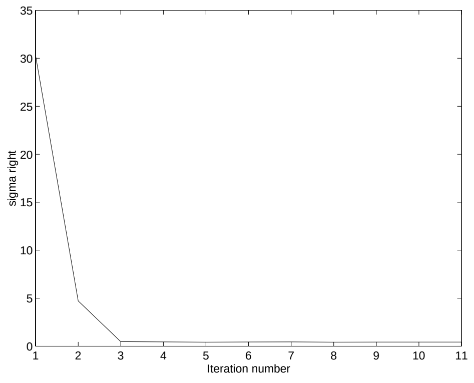


Figure 4.14: σ_r verus the number of iterations in the ELBG case

4.6 ELBG overhead estimation

The aim of this Section is to evaluate the ELBG overhead with respect to classical LBG.

We used the same data set of the previous section and, in all tests performed, we fixed $\epsilon = 0.001$. As the performance of our method depends on the number of codewords, we analyzed several cases ranging from $N_C = 128$ to 1024.

For each dimension of the codebook, we randomly generated 15 initial codebooks. Then, for each codebook a LBG and an ELBG quantization were performed. So, all the reported results are the average of the 15 runs. All runs were executed on a pentium 100MHz based machine and are expressed in seconds.

	LBG	ELBG
128	11.3	12.1
256	22.8	23.7
512	45.3	46.8
1024	91.1	94.3

Table 4.1: Execution times in seconds per iteration

Table 4.1 reports the mean time required for LBG and ELBG. This mean does not comprise the initialization phases.

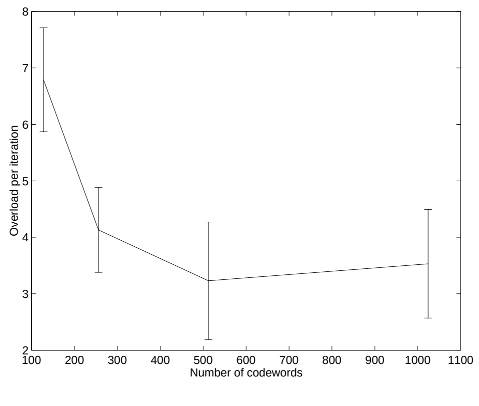


Figure 4.15: Confidence intervals of the percentage time increase per iteration with a confidence level of 99%

In Fig. 4.15 we report the confidence intervals of the percentage time increase per iteration with a confidence level of 99%. This figure shows that the overhead we have introduced in the LBG algorithm is very low. Further, when the number of codewords increases, the overhead decreases below 5%.

N_C	LBG_{rnd}	LBG_{spl}	ELBG_{rnd}	ELBG_{spl}
128	23.6	13.4	11.4	11.2
256	19.4	12.0	11.0	10.4
512	19.2	10.8	10.8	10.6
1024	19.8	10.0	15.4	11.8

Table 4.2: Number of required iterations

Table 4.2 shows the average number (5 runs) of iterations required respectively for the LBG with random initialization (LBG_{rnd}), the LBG with initialization by splitting (LBG_{spl}), the ELBG with random initialization (ELBG_{rnd}) and the ELBG with initialization by splitting (LBG_{spl}). The results reported show that the LBG_{rnd} is the worst of all. Up to a codebook of 512 codewords, both the ELBG_{rnd} and the ELBG_{spl} require less iterations than the LBG_{spl}. The results we obtain for a codebook with a size of 1024 seems to show that the ELBG works worse than the LBG. In effect, more iterations are required. But the reason is that the LBG stops in local minimums. Viceversa, the ELBG succeeds in escaping from these minimums better, and consequently, it requires more iterations.

Fig. 4.16 shows the result we obtain in the case of a codebook with 1024 codewords. The dashed line refers to the LBG_{spl} whereas the other to the ELBG_{rnd}. After only three iterations the ELBG reaches a RMSE equal to about 20.0. When the LBG stops it reaches a RMSE equal to about 20.7. Of course the LBG_{rnd} performs worse.

4.7 Results and comparisons

In this Section we will examine the ELBG performance with several application examples ranging from simple bidimensional quantization approaches to complex image compression tasks. We will compare our results with the most recent results we have found in literature. Also in these examples we fixed $\epsilon = 0.001$.

4.7.1 Qauntization of bi-dimensional cases

In [59] the authors present a new technique and examine several bidimensional cases. We examined two of these.

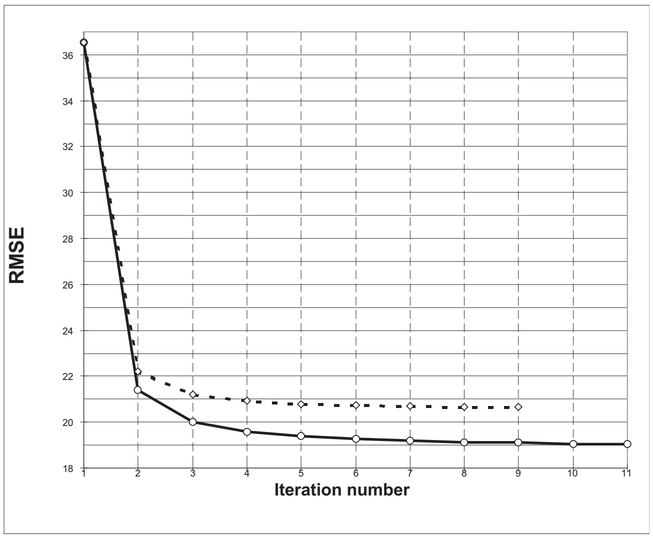


Figure 4.16: RMSE versus number of iterations. ELBG$_{\text{rnd}}$ (solid line) and LBG$_{\text{spl}}$ (dashed line)

Polynomial case: as first case study we have taken 2000 patterns as follows:

$$\begin{cases} x_1 \in [-0.5, 0.5] \\ x_2 = 8x_1^3 - 3x_1 \end{cases} \tag{4.10}$$

where the x_1 values are uniformly spaced in their interval. We fixed $N_C = 16$. To improve statistical accuracy, we have averaged the simulation results over 5 runs with different initial codebooks.

	ELBG$_{\text{rnd}}$	ELBG$_{\text{spl}}$	LBG$_{\text{rnd}}$	LBG$_{\text{spl}}$	[59]
Iter.	10.4	10.6	14.8	7.8	
NMSE	1.1E-2	1.1E-2	2.9E-2	1.1E-2	$\sim 1.1E-2$

Table 4.3: $x_2 = 8x_1^3 - 3x_1$

We have obtained the results reported in Table 4.3. All methods substantially reach the same result except the LBG with random initialization.

Fig. 4.17 shows an initial distribution. Figs. 4.18 and 4.19 show the results obtained respectively with the LBG and ELBG algorithms starting from this initial distribution. The LBG leaves half of the codewords unused, and, for this reason, it gives the worst performance of all, whereas the

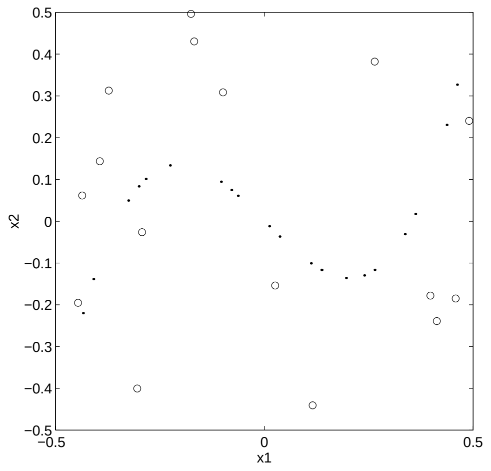


Figure 4.17: Initial distribution

ELBG uses all 16 codewords. In this case the differences between the various methods are marginal because all methods (except the LBG with random initialization), probably, find the global minimum.

Cantor distribution: as second bidimensional case, we examined the three-level Cantor distribution [59]. Even in this case we considered 2000 patterns and 16 codewords. Our results, averaged on 5 runs, are reported in Table 4.4

	ELBG_{rnd}	ELBG_{spl}	LBG_{rnd}	LBG_{spl}	[59]
Iter.	5.2	4.6	5.8	4.6	
NMSE	1.2E-2	1.2E-2	3.2E-2	1.2E-2	$\sim 1.2E-2$

Table 4.4: Cantor's distribution

Fig. 4.20 shows the typical final situation when the LBG algorithm is used. In this Figure the points represent the data set whereas the circles are the codewords. Fig. 4.21 shows one of the five runs we needed to obtain the average reported in Table 4.4. We can see that all codewords are nearly all well positioned in only three iterations and in the next one the process ends.

Also in this case the differences between the various methods are marginal because, probably, all methods (except the LBG with random initialization) find the global minimum or a value very near to it.

Fritzke comparison: in [38] another bidimensional data-set was used by Fritzke to compare his method, called LBG-U, with the standard LBG.

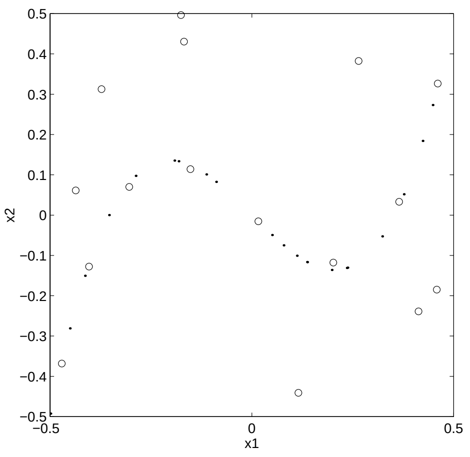


Figure 4.18: LBG final distribution

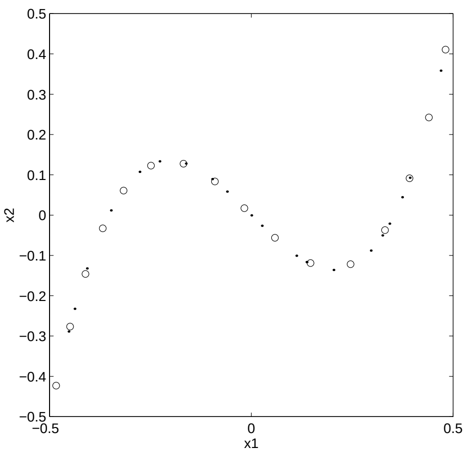


Figure 4.19: ELBG final distribution

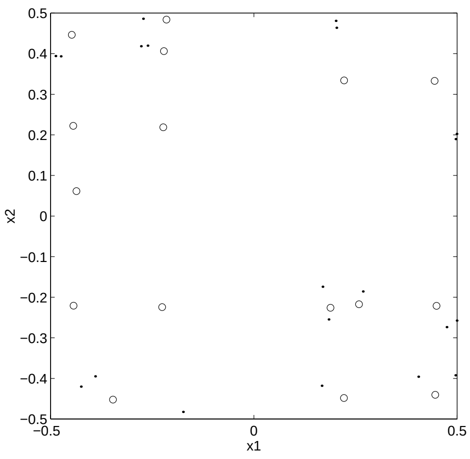


Figure 4.20: LBG final distribution

The experiments were done on a set of 500 learning patterns[2] generated by a Gaussian mixture distribution. The author performed several runs with N_C ranging from 10 to 100. For all codebook sizes the mean improvement of Fritzke's algorithm was higher than 10% with respect to the LBG. But the LBG-U requires a lot of iterations. This number goes from three to seven times that of the LBG method.

We have performed the same tests and averaged the results of 10 runs. In Table 4.5 we have reported our results and the previous ones. It is evi-

[2]available from ftp://ftp.neuroinformatik.ruhr-uni-bochum.de/pub/data/LBG-U.dat

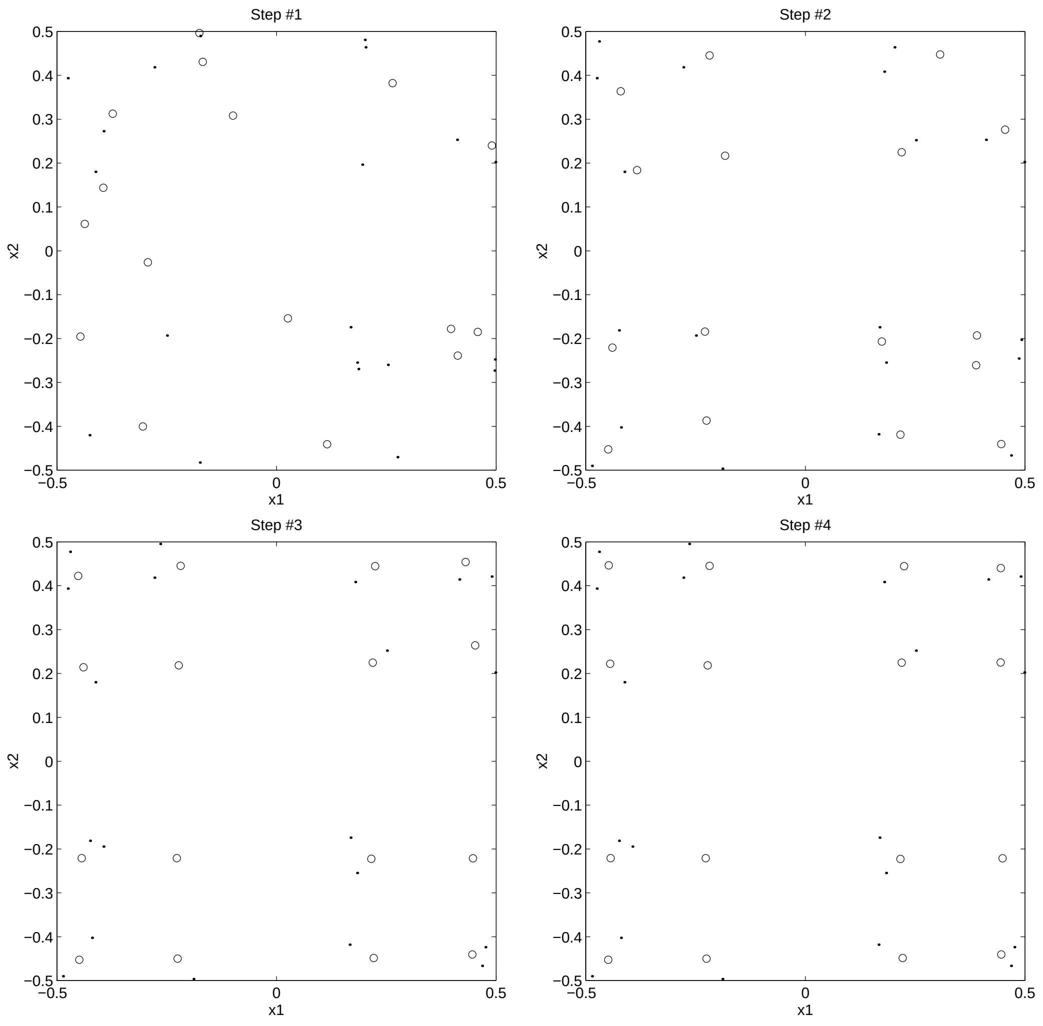


Figure 4.21: ELBG data set and codewords adjustment

N_C	LBG-U		ELBG	
	RMSE $\pm\sigma$	Iter.	RMSE $\pm\sigma$	Iter.
10	$0.0453 \pm 12.5\%$	$31.5 \pm 42.7\%$	$0.0433 \pm 0.20\%$	$7.4 \pm 28.0\%$
100	$0.0125 \pm 2.1\%$	$57.7 \pm 22.1\%$	$0.0123 \pm 0.41\%$	$10.6 \pm 10.8\%$

Table 4.5: ELBG and LBG-U comparison

dent that ELBG outperforms LBG-U both as regards the final error and the number of required iterations. We need about 20% of the iterations required with the LBG-U method.

4.7.2 Image compression

In image applications, often, the Peak Signal to Noise Ratio (PSNR) is used to evaluate the resulting images after the quantization process. The PSNR is defined as follows:

$$\mathrm{PSNR} = 10\log_{10}\frac{255^2}{\frac{1}{IJ}\sum_{i=0}^{I-1}\sum_{j=0}^{J-1}(f(i,j)-\hat{f}(i,j))^2} \tag{4.11}$$

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 $[0,255]$.

Comparison with previous works: Fig. 4.22 shows the original image of Lena (512×512 pixels). Figs. 4.23 and 4.24 show the encoded images using 32 codewords respectively with LBG and ELBG with random initialization. Comparing these two figures, we find an obvious improvement.

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 4.6: Lee et al. and ELBG comparison

In [39] Lee et al. present an enhanced performance K-means algorithm. They improved both the classical K-means algorithm and Jancey's method

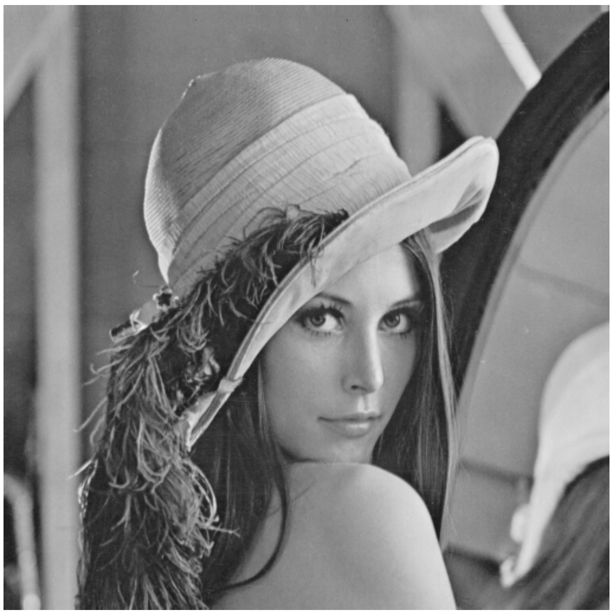


Figure 4.22: Original image of Lena

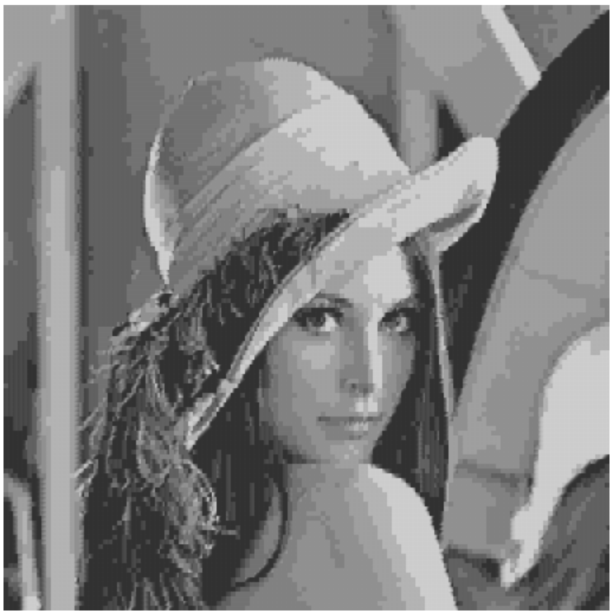


Figure 4.23: LBG reconstructed image of Lena

[40]. The modified K-means algorithm was stopped when the MSE was

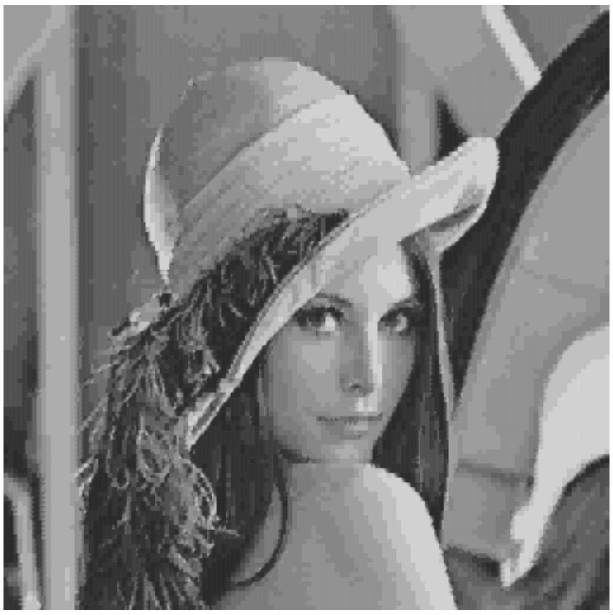


Figure 4.24: ELBG reconstructed image of Lena

within 0.05% of the previous one. This is equivalent to our $\epsilon = 0.00025$. Table 4.6 shows previous results and our present results when initialization by splitting is used. These results refer to Lena's image 512×512 pixels of 8-bit gray levels. We underline that our results are averaged on 5 runs. In this case we improved both the error and the total number of iterations, that is about half.

In [53] Karayiannis and Pai improve Fuzzy Algorithms for Learning Vector Quantization. They used the Lena image of size 256×256 pixels of 8-bit gray values and 512 codewords. As their method depends on several parameters, they executed several runs with different parameter values. Among their results the best one was a PSNR of 32.62 dB. With the same initial hypothesis we obtained 33.04 dB with the random initialization and 33.09 dB with the initialization by splitting. These results are each the average of 5 runs.

A study when N_C increases. When N_C increases the quantization problem becomes more difficult. In fact, more codewords and, consequently, more parameters must be found. Therefore, more local minima can be found by the VQ. We think that a good VQ should work well even in this cases. In Fig. 4.25 we report the results we obtained. We used Lena's image with 512×512 pixels. They are the average PSNR calculated on 5 runs. The worst performance is obtained by the LBG algorithm both with random

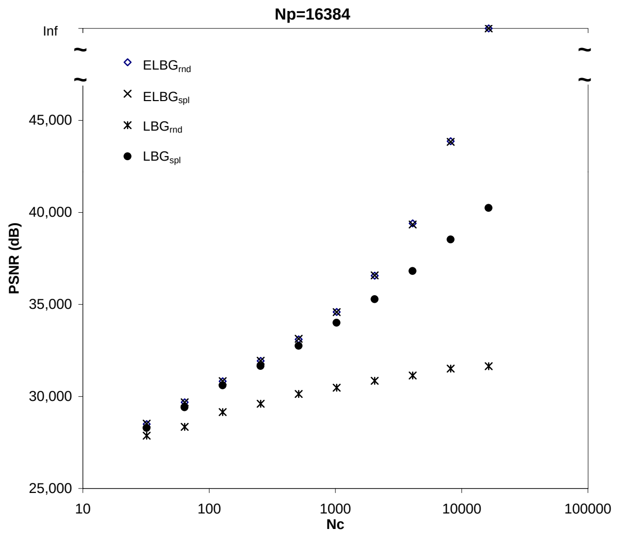


Figure 4.25: PSNR in dB versus the size of the codebook

initialization and initialization by splitting. The figure clearly shows that LBG with random initialization finds very bad local minimums. And when N_C increases there is very little improvement in the PSNR. With initialization by splitting things go better, but a comparison with the ELBG shows that, above all for N_C greater than 4096, there is a large difference. ELBG succeeds in escaping from bad local minimums. Further, when $N_C = N_P = 16384$, it finds the global optimum. In a few iterations it puts every codeword equal to a different learning pattern!

4.8 Conclusions

In this chapter a new clustering technique, called ELBG, was introduced. It is based on the concept of utility of a codeword. This new quantity shows very interesting properties. It allows us to understand which codewords are badly positioned and where they should be moved to escape from the proximity of a local minimum in the error function. The analysis of the main properties of the utility index has permitted us to develop an algorithm whose computational complexity is negligible in comparison to the simpler LBG algorithm. This algorithm improves decidedly the performance of the works regarding the most recent advances in clustering tasks. Further, ELBG shows all its potentiality when the number of codewords increases. As the number of parameters to be found (the components of all codewords) increases, the error function becomes more and more complex and there are plenty of local minimums. So, it becomes very difficult to reach good results. With a significant example we have shown ELBG works well also in these cases. Our results have highlighted that when the number of codewords increases ELBG improvement increases, too.

Chapter 5

The Enhanced LBG Implementation

The aim of this chapter is to describe the particular solutions adopted to keep the overhead of ELBG low with respect to traditional LBG [4]. 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 chapter is organized as follows: first, the symbols used are described; afterwards, LBG and a possible implementation of it are presented; therefore, we describe ELBG and its implementation. Lastly, some results are presented.

5.1 Notation

Before we begin the description of our implementation of 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.

5.1.1 Terminology

In the rest of the chapter, 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 cell i" with the shorter term "i-pattern".

5.1.2 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 tri-dimensional 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.

5.1.3 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 $A3$.

5.1.4 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 $false$.

5.1.5 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.

5.2 LBG 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.(2.4). Several techniques for calculating an initial codebook exist [32] but, in this work, 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
// $\epsilon$ 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
   $C = S./NPC;$
   }
```

5.3 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 (chapter 4.4).

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 (4.2) (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).

5.3.1 Rearrangement of the patterns

The execution of the SoCAs inside the ELBG block 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.
In Fig. 5.1 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. 5.2.
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

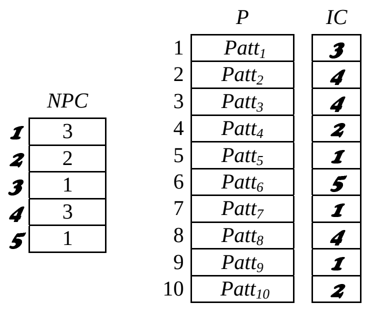


Figure 5.1: An example with 10 patterns and 5 codewords. This is the situation of the matrices P, IC and NPC after the calculation of the Voronoi partition.

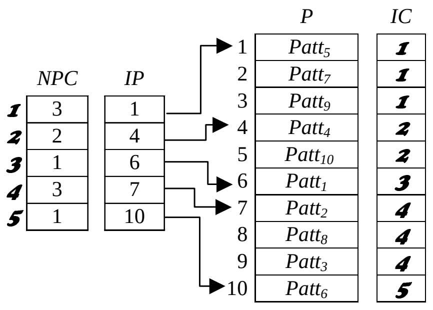


Figure 5.2: The same matrices of Fig. 5.1 are reported after the rearrangement proposed. We can see that all of the patterns belonging to the same cell are stored in consecutive rows of P. The vector IP is reported, too.

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 $IP[i]$.

5.3.2 The technique employed for the rearrangement

As for the rearrangement of the data, we tried to reduce the number of computer memory accesses and data movements. For this reason, we minimize the number of shiftings of the patterns by operating with their indices. We

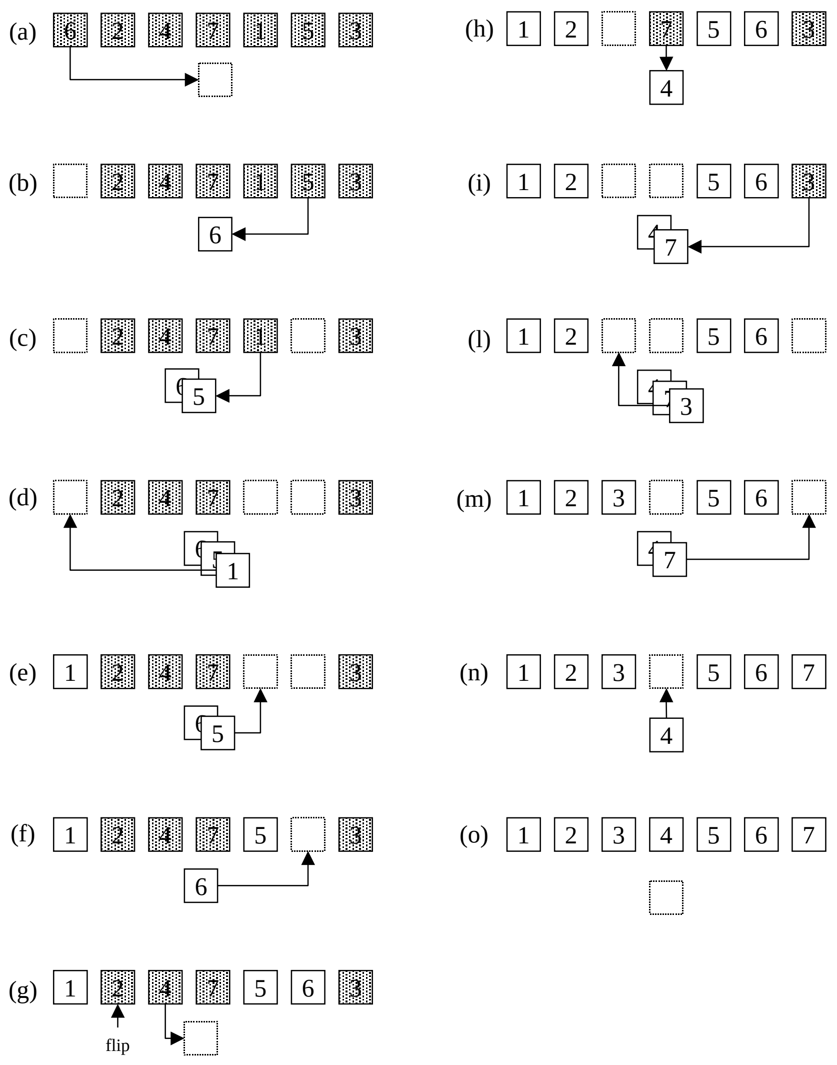


Figure 5.3: Example of rearrangement.

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 complete procedure we followed will be given in section 5.5.

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. 5.3(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. 5.3(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. 5.3. 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 sub-figure (o) where all of the cards are sorted in an increasing order.

In section 5.5 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. 5.1 into that of Fig. 5.2.

5.3.3 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 stored. 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.

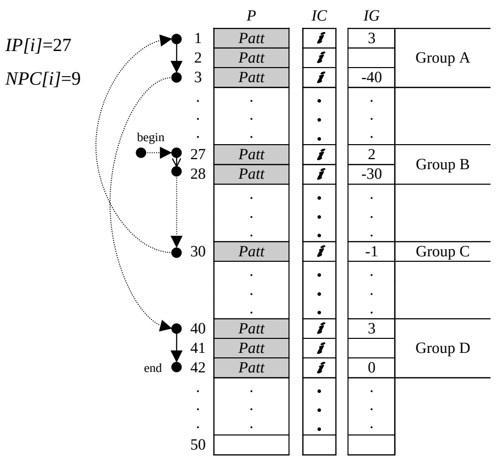


Figure 5.4: Access to the patterns of the generic ith cell when they are, for example, distributed among 4 fragments.

In Fig. 5.4 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. 5.4), 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. 5.4 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. 5.1 into the one of Fig. 5.2. In this case, IG can be obtained from NPC and IP, as the patterns are sorted according to increasing values of the field IC.

5.3.4 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 5.1 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.

5.4 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.

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 5.1: Matrix and vectors adopted to store the data

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.

5.4.1 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 $P[j,:]$;
  $S[i,:] += P[j,:];$
  $NPC[i]++;$
  $D[i] += d(P[j,:], C[i,:]);$
  $D_n += d(P[j,:], C[i,:]);$
  $for(r = 1; r <= K; r++)$
    {$H[i,r,1] = min(H[i,r,0], P[j,r]);$
    $H[i,r,2] = max(H[i,r,1], P[j,r]);$
    }
  $IC[j] = i;$
  }
```

5.4.2 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. 5.1 to the sorted
// form of Fig. 5.2
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 5.3.3;

$for(i = 1; i <= N_C; i++)$
  {$D_{\mathrm{mean}} = D_m/N_C$;
  $if((D[i] < D_{\mathrm{mean}})\ AND\ (Sp[i] == false))$
    {// 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
        // (4.8) and (4.9), 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 by-passed 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.

5.4.3 Looking for cell S_p

S_p is searched by the roulette-wheel method (4.3). Now we will describe the complete procedure to select S_p; we must remember that codewords deriving from unions cannot be split.

Roulette_wheel()

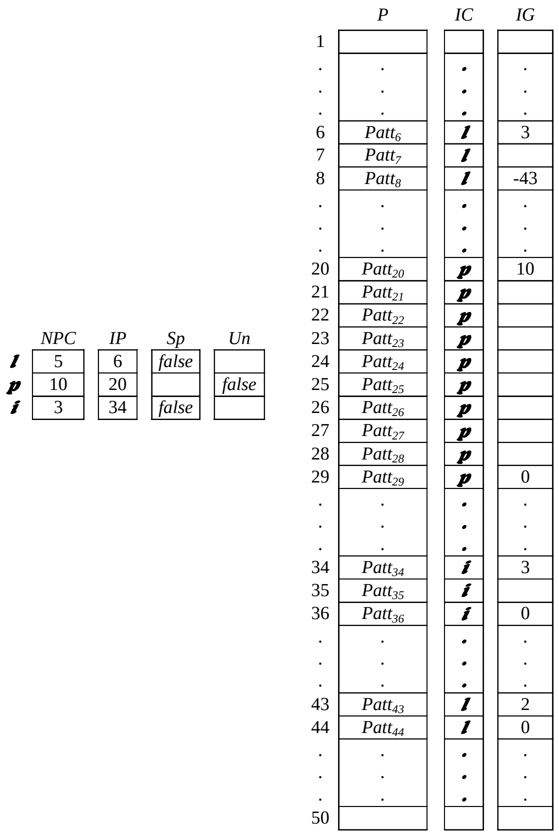
```
// Let us calculate and store in $x$ the sum
// of the distorsions of all the cells that can
// be split.
$for(x = 0, p = 1; p <= N_C; p++)$
  $\{if(D[p] > D_{\text{mean}}\ AND\ Un[p] == false)$
    $x += D[p];$
  $\}$

// 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 (4.3). First, let us
// generate a uniformly distributed
// random number $y$ in the range $[0, x]$
$y = random(x);$
$for(x = 0, p = 1; p <= N_C; p++)$
  $\{if(D[p] > D_{\text{mean}}\ AND\ Un[p] == false)$
    $\{x += D[p];$
    $if(x >= y)$ // this value of $p$ is returned
      $return\ \ p$ // the procedure ends
    $\}$
  $\}$
```

5.4.4 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			

Figure 5.5: 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. 5.5. 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 4.4.3. 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. 5.5). In 4.4.3, 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. 5.6.

5.4.5 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 5.3.1 and 5.3.3. The schematic description of the SoC is reported here; after we will explain it.

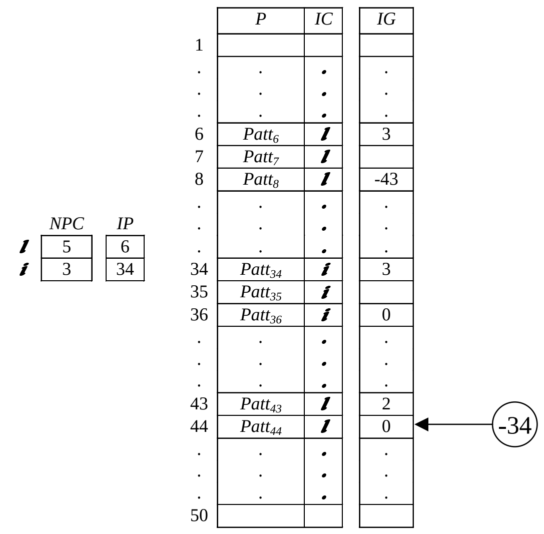


Figure 5.6: Linking of the last l-pattern to the first i-pattern. This operation will be executed during the SoC.

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. 5.6. 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. 5.7. 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

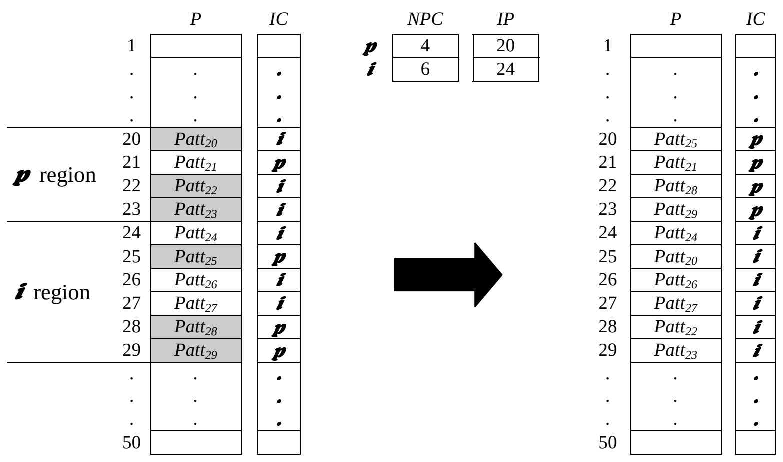


Figure 5.7: Local rearrangement of the patterns.

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.

After the SoC, we have the situation of Fig. 5.8 and it has to be compared with that of Fig. 5.5.

5.5 Detailed description of the technique employed for the rearrangement

Now we will describe the procedure we execute to lead the data from the unsorted form of Fig. 5.1 to the sorted situation of Fig. 5.2. This technique derives from the one described in 5.3.2 and illustrated in Fig. 5.3. 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

	NPC	*IP*	*Sp*	*Un*
l	8	6	*false*	*true*
p	4	20	*true*	*false*
i	6	24	*true*	*false*

	P	*IC*	*IG*
1			
.	.	.	.
.	.	.	.
.	.	.	.
6	*Patt*$_6$	***l***	3
7	*Patt*$_7$	***l***	
8	*Patt*$_8$	***l***	-43
.	.	.	.
.	.	.	.
.	.	.	.
20	*Patt*$_{25}$	***p***	4
21	*Patt*$_{21}$	***p***	
22	*Patt*$_{28}$	***p***	
23	*Patt*$_{29}$	***p***	0
24	*Patt*$_{24}$	***i***	6
25	*Patt*$_{20}$	***i***	
26	*Patt*$_{26}$	***i***	
27	*Patt*$_{27}$	***i***	
28	*Patt*$_{22}$	***i***	
29	*Patt*$_{23}$	***i***	0
.	.	.	.
.	.	.	.
.	.	.	.
34	*Patt*$_{34}$	***l***	3
35	*Patt*$_{35}$	***l***	
36	*Patt*$_{36}$	***l***	0
.	.	.	.
.	.	.	.
.	.	.	.
43	*Patt*$_{43}$	***l***	2
44	*Patt*$_{44}$	***l***	-34
.	.	.	.
.	.	.	.
.	.	.	.
50			

Figure 5.8: Situation of the data related to the cells i, l, p, after the SoCA

that are already in their correct locations are never moved. The technique of Fig. 5.3 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. 5.1. There, an input data set of 10 patterns belonging to 5 different cells is represented. From this figure, let us construct Fig. 5.9.

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 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. 5.9 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

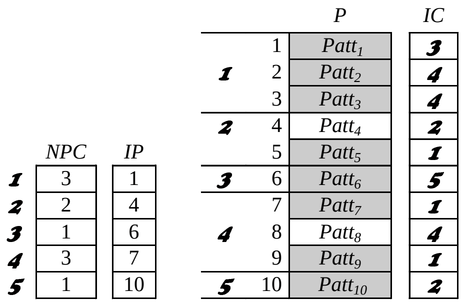


Figure 5.9: Initial situation. Only the arrays and the matrices involved in the sorting operation are reported.

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 5.3, in a configuration like that of Fig. 5.2, 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. 5.9. 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. 5.10 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. 5.10.

- 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

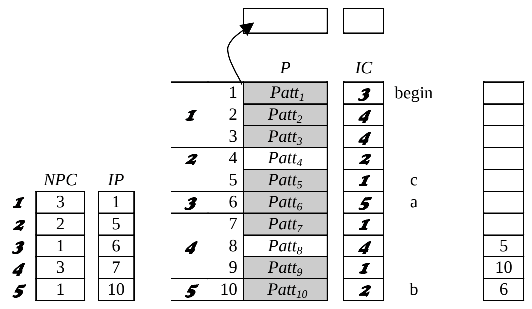


Figure 5.10: Sorting: stack creation.

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. 5.11.

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 stack and, iteratively, repeat this procedure until the stack is empty. (Fig. 5.11-5.13). Afterwards, the pattern in the temporary location is shifted into the empty row of P (Fig. 5.13) and $IP[1]$ is increased by one.

After the first sequence of operations, we have a situation as in Fig. 5.14. The procedure just described starts again from row 2 and continues until all the vector is sorted as we wish.

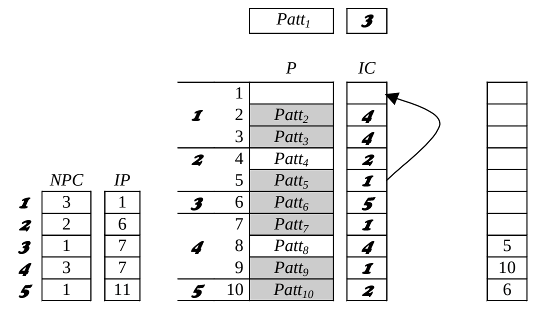


Figure 5.11: Emptying of the stack (I).

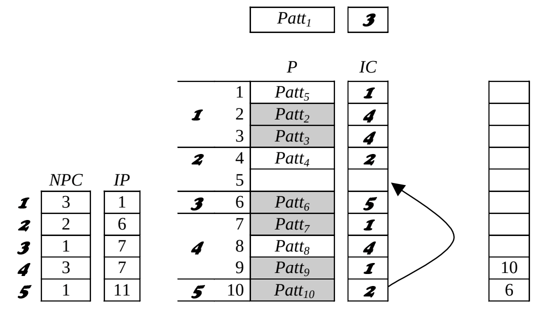


Figure 5.12: Emptying of the stack (II).

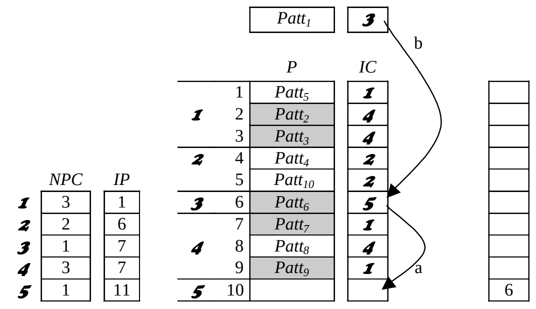


Figure 5.13: Emptying of the stack (III).

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. 5.3, 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.

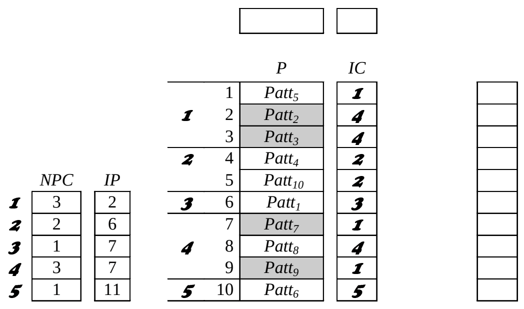


Figure 5.14: Situation after the completion of the first sequence of operations.

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. 5.2.

Chapter 6

Fully Automatic Clustering System (FACS)

6.1 Introduction

In this chapter we describe the Fully Automatic Clustering System (FACS), a CA/VQ technique whose objective is to automatically find the codebook of the right dimension, when the input data set, the distortion measure and the desired error (or target, e_T) are fixed [5]. It develops through a sequence of iterations like ELBG, that is taken as a starting point. But, unlike ELBG, FACS, evaluates the error at the end of each iteration and, according to whether it is above or below the target, decides to make another iteration with the same number of codewords or, if it is necessary, to increase or decrease N_C. Such an increase or decrease happens *smartly*, trying to insert new codewords where the quantization error is higher and to eliminate them where the error is lower. A similar strategy is proposed by Fritzke in [61, 63] for his competitive-learning algorithms, while FACS is a K-means type algorithm [3, 88]. Besides, in FACS, insertions and deletions of codewords are regulated by a stochastic process; in [61, 63] they occur deterministically. Particular attention is paid so that the technique employed allows the convergence of the algorithm towards a good solution in a few iterations. The results presented in section 6.8 will show that a number of iterations comparable with the ones required by ELBG are enough. In chapter 4 the high speed of convergence of ELBG has already been highlighted.

The chapter is organized as follows: section 6.2 reports some considerations about ELBG. They suggest two useful criterions for the deletion and the insertion of codewords into the codebook; in sections 6.3-6.7 FACS is presented in detail and its performances are reported in 6.8. Conclusions are presented in section 6.9.

6.2 Considerations regarding ELBG

In chapter 4, we have seen that a SoC consists of the combined execution of two operations: the splitting of a cell and the union of two other cells. So, N_C remains unchanged because the codeword that is eliminated from one place is inserted into another. For this reason we use the word *shifting* to describe the whole operation. Instead, if we execute only the part related to splitting or only that related to union, the number of codewords will increase or decrease, respectively. In this way we can insert or delete some codewords from a codebook. Such operations can be considered, in first approximation, *fast* and *intelligent* because:

- insertions are effected in the regions where the error is higher and deletions where the error is lower;
- operations are executed locally, i.e. without global reorganizations of the codebook and the partition;
- several insertions or deletions can be effected during the same iteration always working locally.

Insertions and deletions of codewords effected by FACS are realized working in this way, as will be explained in detail in the next section.

6.3 General Description

Each of the iterations through which FACS develops (FACS-iterations) can be summarized as in Fig. 6.1. It can be divided into two parts: in the first one the same operations that have been described in chapter 4 for ELBG are executed. They are: the Voronoi partition calculation, the ELBG-block execution and the calculation of the codebook satisfying the CC. For this reason, such a sequence of operations was grouped into a single block that we called ELBG-iteration. A FACS-iteration is completed by the execution of the last block (the FACS-block) whose functionality is to *smartly* modify, if necessary, the number of the codewords.

As illustrated in Fig. 6.2, we can distinguish two phases during the execution of FACS. Each of them is constituted by a sequence of FACS-iterations like the ones in Fig. 6.1. For each FACS-iteration, the ELBG-iteration

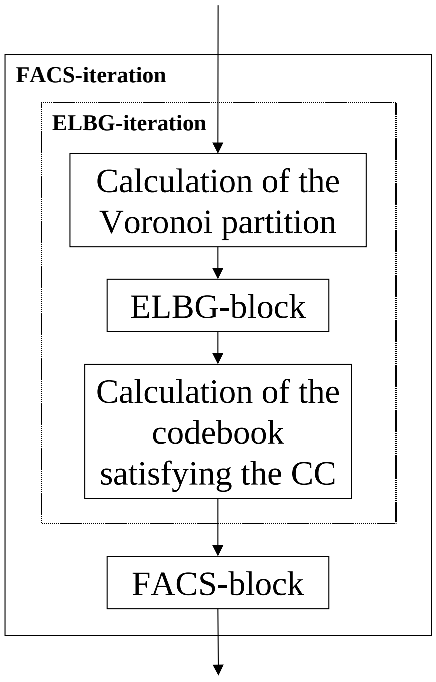

FACS-iteration
ELBG-iteration
Calculation of the Voronoi partition
ELBG-block
Calculation of the codebook satisfying the CC
FACS-block

Figure 6.1: A FACS-iteration

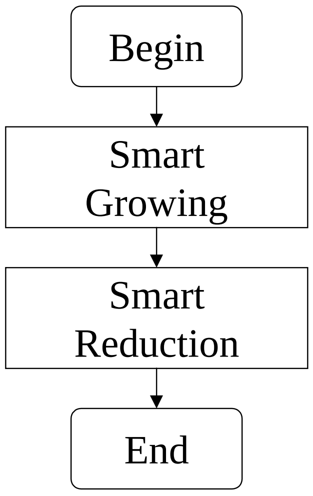


Figure 6.2: The two phases through which FACS develops

executes the same operations for both of the phases, while the operations executed inside the FACS-block are different.

The first phase, called *Smart Growing*, consists of a certain number of FACS-iterations, during which the number of the codewords is gradually increased until the error barely goes below e_T. Such an increase happens by splitting, one at a time, some cells, chosen among the ones with a total distortion greater than D_{mean}, i.e. with U_i greater than 1 (see eqs. (4.1) and (4.2)). So, codewords are inserted where the distortion is higher.

During the second phase, called *Smart Reduction*, a certain number of FACS-iterations are executed during which the number of codewords is decreased. More precisely, if during the first phase of an iteration (the ELBG-iteration) the error goes below e_T, as many codewords as are necessary to obtain $D > e_T$ are deleted. A codeword deletion is realized by choosing a cell whose total distortion is less than D_{mean} (i.e. with U_i less than 1) and by joining it to the nearest cell.

After an insertion or deletion occurs, we can calculate the new distortion by modifying, in (2.4), only the contributions of the cells (D_i) involved in the splitting or in the union. So, we are able to immediately evaluate if other insertions or deletions are necessary without recalculating the Voronoi partition, therefore very quickly. The whole procedure for the selection and the insertion or deletion of the cells will be described later.

Now, let us describe the two phases through which the algorithm develops, as we have shown in Fig. 6.2.

6.4 Smart Growing

The initialization of the codebook could be executed starting from a single codeword in the centroid of the whole input data set and inserting new ones until the error is below e_T. Afterwards, during the next iterations, codewords in excess would be removed. However, we realized, experimentally, that a better result is obtained when the insertion of the codewords is effected more gradually. With the term better result we mean that, under the same value of e_T, final codebooks with lower values of N_C are obtained. The phenomenon was more evident when the complexity of the problem increased.

In Fig. 6.3 the whole phase of the *Smart Growing* is illustrated. The growing happens by inserting, each time, as many codewords as are necessary for the error to be equal to or less than $(1 + p)e_T$, where $p \geq 0$ and is monotonically not increasing from p_{ini} to p_{end} ($p_{end} = 0$). In the same picture, the FACS-block and the operations executed by it when we are in the *Smart Growing* phase are highlighted. Inside the FACS-block, the block dealing with the insertion of the codewords is further highlighted. A more detailed description of its function will be given later in this sub-section. Once the

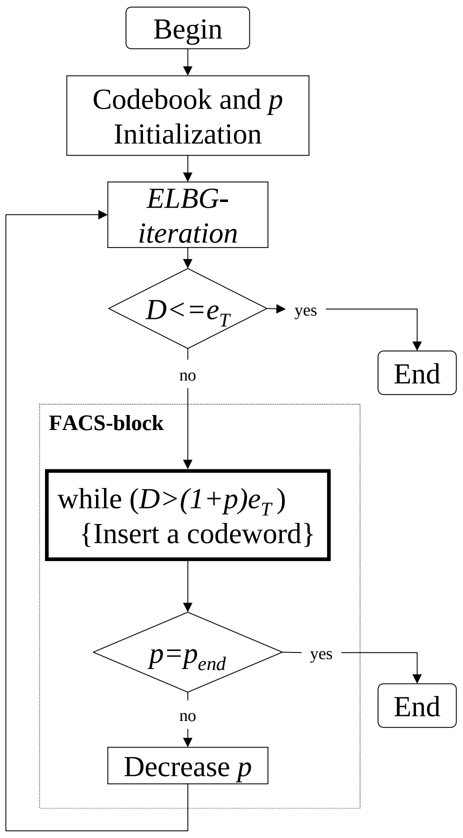


Figure 6.3: The *Smart Growing* phase

law regulating the decrease of p with the iteration number (n) has been fixed, we can summarize the *Smart Growing* as follows.

1. $N_C = 1$ and $n = 1$ being fixed, place the first codeword in the centroid of the whole input data set;
2. insert as many codewords as are necessary to obtain $D \leq (1 + p(n))e_T$;
3. if $(p(n) = 0)$ then the *Smart Growing* ends;
4. execute an *ELBG-iteration*;
5. if $(D \leq e_T)$ then the *Smart Growing* ends;
6. $n + +$;
7. return to point 2.

We verified experimentally that it is better to select a high initial value (about 1) for p and make it decrease quickly iteration by iteration. For this reason we chose to make p decrease from p_{ini} to $p_{end} = 0$ in n_I iterations and that, for the first $n_I - 1$ iterations, it follows an exponential law, i.e.:

$$p(n) = \begin{cases} \alpha e^{-\beta n} & \text{for } n = 1, 2, \cdots n_I - 1 \\ 0 & \text{for } n = n_I \end{cases} \tag{6.1}$$

where α and β are positive constant values.

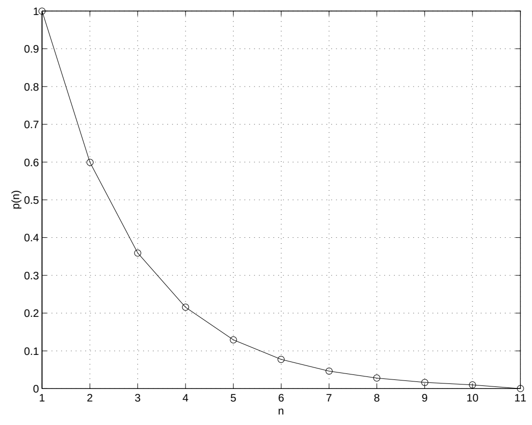


Figure 6.4: p versus the number of iterations

Still experimentally, we saw that about ten iterations are enough to obtain good results. In Fig. 6.4 $p(n)$ is reported when $n_I = 11$ and α and β were fixed so that $p(1) = 1.0$ and $p(n_I - 1) = 0.01$. Such values were used for all of the tests we effected with FACS.

Fig. 6.5 details the operations that are executed by the block regulating the insertion of the codewords inside the FACS-block and highlighted in Fig. 6.3 for the *Smart Growing* phase. If the quantization error is above the desired threshold $((1 + p)e_T))$, the FACS-block inserts the number of codewords necessary to take it just below. This happens by executing several times the procedure of the splitting that was explained in 4.4.3 for a SoCA. For a better understanding of the operations reported in Fig. 6.5, we must keep in mind that:

- The selection of the cell to split and of the related codeword (S_i,$\mathbf{y}_i$) is made among the ones whose utility is greater than 1 with the roulette wheel method (see (4.3)).

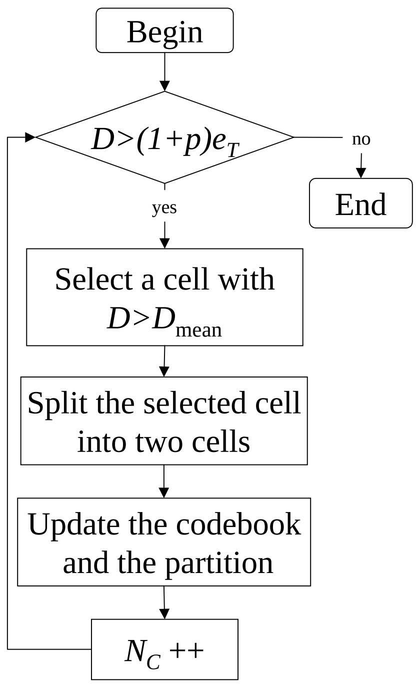

Begin
D>(1+p)e_T
no
yes
End
Select a cell with D>D_mean
Split the selected cell into two cells
Update the codebook and the partition
N_C ++

Figure 6.5: Detailed description of the insertion of the codewords

- The splitting of $(S_i,\mathbf{y}_i)$ in $(S_i',\mathbf{y}_i')$ and $(S_i'',\mathbf{y}_i'')$ happens as is explained in 4.4.3.
- The update of the partition and of the codebook consists of the substitution of $(S_i,\mathbf{y}_i)$ with $(S_i',\mathbf{y}_i')$ and $(S_i'',\mathbf{y}_i'')$.

6.4.1 Discussion about the law regulating the decrease of the target error

The employment of the greedy strategy described above for the insertion of the codewords, allows an enormous saving on the computation. However, it is a non-optimal solution because not all of the elements of the codebook and of the data set are considered. For this reason, each iteration includes, as well as a number of local updates, also a global rearrangement. The non-optimal effect of the local updates is more evident particularly in the early iterations when the codebook is very disorganized. In that case, it is necessary to insert a high number of codewords in order to ensure the desired target. But, if the initial target is higher than the one specified by the user and, gradually, approaches it, during the early iterations a lower number of codewords is inserted. Iteration by iteration, because of the global optimization, the codewords distribute themselves better and better; so, the new insertions can occur more exactly.

Besides, also the following phase, i.e. the *Smart Reduction*, benefits from such a way of operating because it will have to deal with the removal of a lower number of exceeding codewords.

6.5 Smart Reduction

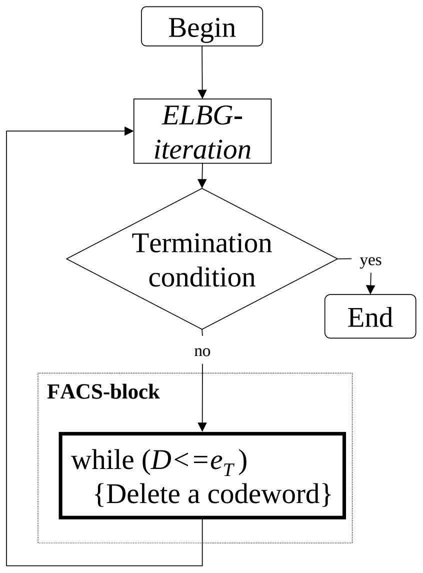


Figure 6.6: The *Smart Reduction* phase

The whole phase of the *Smart Reduction* can be summarized as in Fig. 6.6. The number of the codewords is gradually decreased as soon as, at the end of an iteration, the error is equal to or less than the target. Instead, if the error is above the target we continue with the same resolution, i.e. with the same number of codewords. In Fig. 6.6 the FACS-block and the operations it executes during the *Smart Reduction* phase are highlighted. A more detailed description of it is reported in Fig. 6.7. If the quantization error is equal to or less than the desired value (e_T), the FACS-block deletes the number of codewords necessary to take it just above. The elimination happens in a *smart* way because we try to remove the codewords that contribute the least to the total distortion.

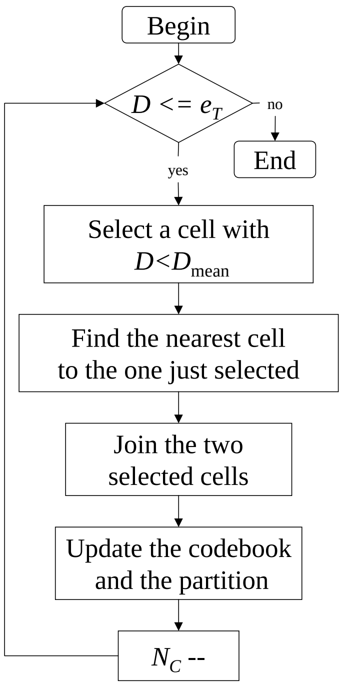

Begin
$D <= e_T$
no
End
yes
Select a cell with $D<D_{\text{mean}}$
Find the nearest cell to the one just selected
Join the two selected cells
Update the codebook and the partition
N_C --

Figure 6.7: Detailed description of the deletion of the codewords

To understand better the operations reported in Fig. 6.7, we must keep in mind that:

- the cell to eliminate and the related codeword (S_i,$\mathbf{y}_i$) are selected among the ones whose utility is less than 1 with a probabilistic method analogous to the one expressed by (4.3). Here, the cell to eliminate is chosen with a probability that is a decreasing function of its distortion. In mathematical terms:

$$P_p = \frac{1 - U_p}{\sum_{h:U_h<1}(1 - U_h)} \tag{6.2}$$

- the union of (S_i,$\mathbf{y}_i$) and (S_l,$\mathbf{y}_l$) to form (S_l',$\mathbf{y}_l'$) is effected as explained in 4.4.3;

- The update of the partition and the codebook consist of the substitution of (S_i,$\mathbf{y}_i$) and (S_l,$\mathbf{y}_l$) with (S_l',$\mathbf{y}_l'$).

The block related to the termination condition will be explained later.

6.6 Behaviour of FACS versus the number of iterations and termination condition

The algorithm was developed so that, during the *Smart Deletion*, the error is always near e_T. This happens thanks to the continuous adjustments of N_C.

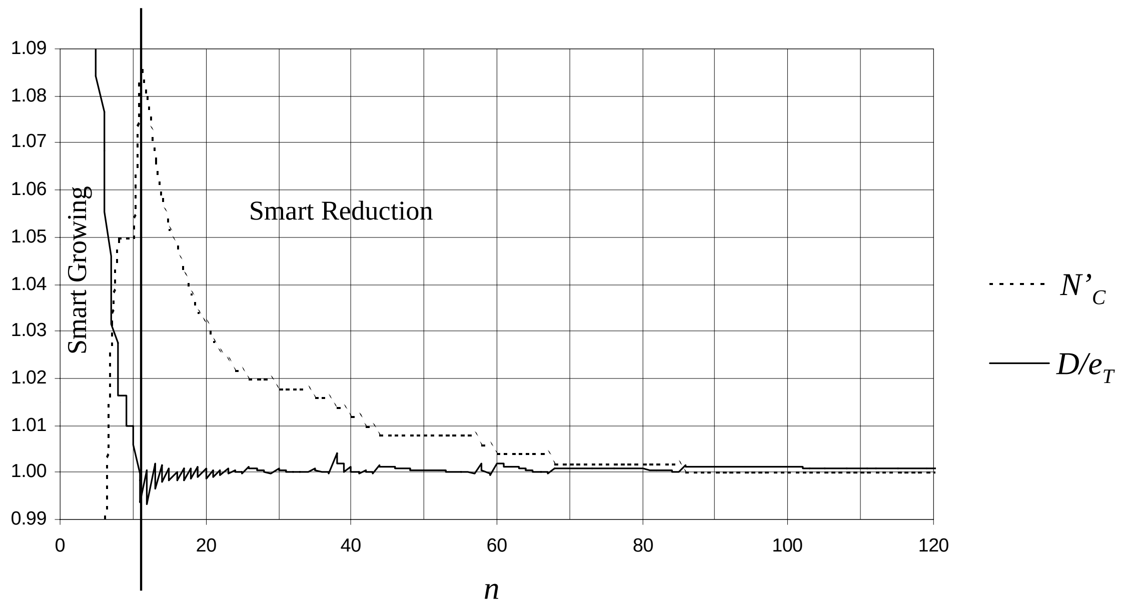


Figure 6.8: Typical trend of N'_C (it is N_C normalized with respect to its value after 200 iterations) and D (normalized with respect to e_T) versus the number of iterations

In Fig. 6.8 we report the typical trend of the error and the number of codewords FACS works with, versus the number of iterations. The graph refers to an image compression task, whose details, that are not, at present, important for understanding the picture, are given in the section related to the comparisons.

For a better graphic visualization, we chose to report the normalized values of the variables in question. In particular, N'_C is N_C normalized with respect to the value found by FACS for that run after 200 iterations; the RMSE (D) is normalized with respect to the target (e_T). We can immediately see that, after the insertion of the codewords ends, the algorithm always keeps the error very close to e_T. In particular, we can notice the correlation between the two curves represented. When the error is greater than e_T, N_C is kept constant. As soon as D goes below e_T, N_C is automatically decreased until the error is above e_T.

The graph just examined suggests two criteria to employ as termination conditions for FACS. Both of them are to be adopted when the *Smart Growing* phase has ended.

- the algorithm ends after a certain number of prefixed iterations;
- the algorithm ends when a certain number of consecutive iterations with the same value of N_C have been executed.

If we choose one of them as the termination condition, it is possible that, when FACS ends, the quantizer obtained, with N_C codewords, generates an error greater than e_T. Such an error, considering how the algorithm develops, is very close to e_T. So, according to its value, it could be considered acceptable. However, even if it cannot be considered acceptable, it is sufficient to remember that the quantizer with N_C codewords was obtained by eliminating a codeword from the one with $N_C + 1$ codewords, that produced an error less than e_T. Therefore, if we keep in memory the last codebook that was able to ensure an error less than e_T, we can stop the algorithm at any moment and have a codebook satisfying the desired specifications.

6.7 Discussion about outliers

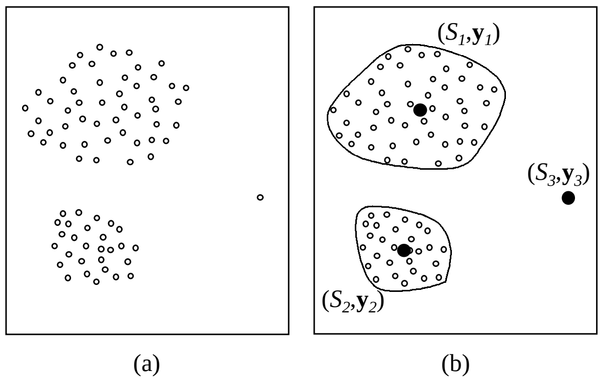


Figure 6.9: Dataset with two clusters and one outlier

We wish to underline again that FACS has been conceived with the aim of autonomously calculating an opportune codebook having, as its only requirement, the satisfying of a target error (specified by the user) with the least number of codewords. However, the presence of outliers may degrade the results obtained by FACS. For example, let us consider Fig. 6.9(a). There, we can locate two clusters of points and one outlier point rather "far" from the clusters. Let us suppose FACS is launched and that, for a certain value of e_T, it finds 3 codewords, one in the center of each cluster and one coinciding with the outlier, as in Fig. 6.9(b). According to eq. (2.4), the MQE is given by:

$$\text{MQE} = \frac{D_1 + D_2 + D_3}{N_P} \tag{6.3}$$

Given that the outlier is exactly represented by codeword number 3, we have $D_3 = 0$. Now, if we calculate the MQE only on the "clean" part of the data set, i.e. excluding the outlier, it is:

$$\text{MQE}_{clean} = \frac{D_1 + D_2}{N_P - 1} \tag{6.4}$$

Obviously, $\text{MQE}_{clean} \geq \text{MQE}$. This implies that the presence of outliers can also heavily affect the final result obtained by FACS. This is not a drawback of FACS, but it derives from the nature of the algorithm itself.

However, there are applications where the identification of the outliers is essential in order to avoid their influence on the final result. In such cases, FACS can still be used, provided a noise category removal mechanism [69] is added for treating the outliers. Even if it is outside the scope of this work, we wish to cite some methods commonly adopted in literature that, opportunely readapted, could represent the desired modifications. For example, a simple mechanism lets the algorithm run with the desired value of e_T until it ends. Afterwards, the outliers are identified (for example, the patterns constituting a cell with a single pattern), and removed (together with the related codewords). Therefore, the algorithm can go on with the "clean" data set so obtained. More advanced mechanisms could provide for the removal of patterns constituting cells whose cardinality is less than a prefixed threshold (for example, a certain quota of the total number of patterns).

6.8 Results and comparisons

6.8.1 Introduction

In this section we present the results obtained by FACS and compare it with both algorithms where the codebook has a fixed size, and techniques where the dimension of the codebook is not known.

In order to perform the comparisons, both in terms of final result and speed of convergence, we looked for similar techniques already documented in literature. We found several algorithms that, as underlined by their authors, can generate a codebook of opportune dimension that is able to ensure the desired level of performance [61–64]. However, we noticed they have been used either as traditional algorithms for VQ/CA [61], where N_C is fixed, or as algorithms for supervised learning and classification [63]. For some of them it is explicitly said that they can be used to generate a codebook of the opportune size once the desired error has been fixed [62, 64]. Unfortunately, we did not find any example where they were employed in this way. Besides, also when they were used as traditional VQ/CA algorithms, we found no numerical results directly comparable with FACS. For example, in [61], Fritzke reports the results related to an image compression task for its Growing and Splitting Elastic Net. In particular, he uses the image of Lena [86] of 480×480 pixels at 256 levels of gray. But, his results also consider an intermediate processing of the data by means of a multi-layer perceptron; so, they are not directly comparable to the ones obtained by FACS.

However, comparisons with some of these algorithms ([64] and GNG-U [89]) have been performed following the suggestion of the author, i.e. making the number of codewords "grow" until the desired target is reached. Besides, the performances of FACS can be compared with the ones of a traditional algorithm for VQ/CA where N_C is a datum. In fact, it is possible to fix a value of N_C and see what error the VQ algorithm obtains. Afterwards, we can choose that error as the target for FACS and see how many codewords are necessary to obtain that performance. The comparison has been executed with ELBG, that has obtained performances better than or equal to the ones of the previous VQ algorithms existing in literature [38, 39, 53, 59].

Several more algorithms exist in literature where the number of the codewords is not a datum but one of the results. Among them we have chosen to make comparisons with FOSART, an algorithm in the family of Adaptive Resonance Theory (ART) [48, 69] and the Competitive Agglomeration algorithm [45].

Afterwards, we have tried using FACS for one of the classification problems reported in [63].

All of the tests have been effected on machines equipped with Intel Celeron 366 MHz processors and running the LINUX operating system. The related details will be given later.

As we have said in 6.7, FACS has been conceived with the aim of autonomously calculating an opportune codebook having, as its only requirement, the satisfying of e_T with the least number of codewords. We are aware that some of the algorithms we used for our comparisons were designed for solving different problems. So, a direct comparison with FACS may appear rather forced. However, the techniques considered are the works in literature that, to the best of our knowledge, can be considered more similar to ours.

6.8.2 Comparison with ELBG

In this sub-section we evaluate the performances of FACS for a typical task of VQ: image compression. For our tests we chose the image of Lena of 512×512 of 256 gray levels. It was subdivided into 16384 blocks of 4×4 pixels and the related 16-dimensional vectors were used as learning patterns.

The procedure employed to effect the comparisons develops through the following points:

1. a value for N_C being fixed, ELBG is executed and its results, in terms of RMSE, are collected;

2. FACS is executed using the value of the RMSE obtained by ELBG as target;

3. the number of codewords found by FACS is compared with N_C.

The comparison was repeated for several values of N_C. As we will see, FACS obtains, practically, the same number of codewords ELBG had been launched with. Besides, we will see that it is possible to fix a relatively low number of iterations (we have chosen 15) and obtain results nearly equal to the ones we would obtain by making FACS run for a much longer period. In the remainder of the article, we describe in detail the procedure we have followed.

The ELBG was launched for $N_C = 128, 512, 4096$ and we chose a very low value for ϵ (0.00000001). In this way, the algorithm ends when, practically, it reaches the minimum for that run.

In table 6.1 the results obtained are reported. All of the values are the mean calculated on 10 runs. For each value of N_C we report the results obtained after 15 iterations and the ones obtained after the algorithm ends (∞ iterations). Each result is reported together with the variance calculated on the runs effected.

N_{it}	$N_C = 128$		$N_C = 512$		$N_C = 4096$	
	RMSE	$t_{it}(s)$	RMSE	$t_{it}(s)$	RMSE	$t_{it}(s)$
15	29.26 ± 0.06	2.38 ± 0.04	22.39 ± 0.03	8.76 ± 0.19	10.81 ± 0.02	106.57 ± 0.56
∞	29.20 ± 0.07	2.25 ± 0.03	22.33 ± 0.03	8.67 ± 0.04	10.74 ± 0.02	106.75 ± 0.56

Table 6.1: ELBG performance

Before we proceed to the comparisons with ELBG, some considerations regarding the calculation times are necessary. As we saw in chapter 4, the time required per iteration, once the input set has been fixed, increases almost linearly when N_C increases, too. As we can see from Table 6.1, this is, more or less, valid when passing from $N_C = 128$ to $N_C = 512$, while this is not true when passing from $N_C = 512$ to $N_C = 4096$. Such behaviour occurs because, in such a complicated task (the ratio $\frac{N_P}{N_C}$ is 4), about half of the patterns are involved in SoCA's during the execution of the ELBG-block. Instead, in the other cases considered, the contribution of the ELBG-block is lower and is in agreement with the rate (about 5%) that we indicated in chapter 4 and the time per iteration increases almost linearly when N_C increases, the other parameters being fixed.

e_T		$N_{it} = 15$	$N_{it} = 200$
RMSE$_{15}$(128)	N_C	128.4 ± 0.8	124.9 ± 0.7
	$\frac{N_C}{128}$	1.003 ± 0.006	0.976 ± 0.005
	N'_C	1.028	1
	$t_{it}(s)$	2.16 ± 0.03	2.14 ± 0.03
RMSE$_\infty$(128)	N_C	130.8 ± 1.6	126.4 ± 1.2
	$\frac{N_C}{128}$	1.022 ± 0.013	0.988 ± 0.009
	N'_C	1.035	1
	$t_{it}(s)$	2.21 ± 0.04	2.17 ± 0.03
RMSE$_{15}$(512)	N_C	517.4 ± 3.0	496.7 ± 2.2
	$\frac{N_C}{512}$	1.011 ± 0.023	0.970 ± 0.017
	N'_C	1.042	1
	$t_{it}(s)$	7.83 ± 0.05	8.29 ± 0.03
RMSE$_\infty$(512)	N_C	527.3 ± 4.3	504.2 ± 4.9
	$\frac{N_C}{512}$	1.030 ± 0.008	0.985 ± 0.010
	N'_C	1.046	1
	$t_{it}(s)$	8.00 ± 0.16	8.41 ± 0.08
RMSE$_{15}$(4096)	N_C	4143.5 ± 16.7	4074.8 ± 16.1
	$\frac{N_C}{4096}$	1.012 ± 0.004	0.995 ± 0.004
	N'_C	1.017	1
	$t_{it}(s)$	100.23 ± 1.38	103.95 ± 1.82
RMSE$_\infty$(4096)	N_C	4188.6 ± 14.9	4118.9 ± 17.3
	$\frac{N_C}{4096}$	1.023 ± 0.004	1.006 ± 0.004
	N'_C	1.017	1
	$t_{it}(s)$	102.39 ± 1.25	105.67 ± 1.29

Table 6.2: FACS performances

Afterwards, FACS was launched with e_T set to the RMSE found by ELBG after 15 iterations ($\text{RMSE}_{15}(N_C)$) and after infinite iterations ($\text{RMSE}_{\infty}(N_C)$) (for the values of N_C considered) as targets. In Table 6.2, the values obtained by FACS after 15 and after 200 iterations are reported. As we can see, FACS obtains codebooks with, practically, the same number of codewords used by ELBG to achieve that error; the difference is inside 3% in all of the considered cases. Besides, the values of N'_C (the number of codewords obtained by FACS normalized with respect to the value found after 200 iterations) show that, after 15 iterations, FACS obtains a number of codewords that is very close to the one it will obtain after 200 iterations; the difference is below 5% in all of the considered cases. Also the mean time per iteration for ELBG and FACS are comparable.

6.8.3 Comparison with GNG and GNG-U

Here, a comparison with GNG and its variant (GNG-U), both from Fritzke [64, 89] is reported. They, gradually, insert codewords until the prefixed number or until a certain "performance measure" is fulfilled. In our case, the performance measure to be considered is the achievement of e_T. The error is calculated at the end of each epoch[1] (or iteration) and, like in FACS, according to whether it is above or below e_T, a decision about the insertion of more codewords is taken.

All of the tests described here have been executed using the image of Lena previously mentioned. Also the simbology is the same as the one used in the previous comparison.

For the execution of GNG, as well as the desired e_T, it is necessary to specify several configuration parameters. In order to choose such values, we have referred to the works from Fritzke where he presented his algorithms [64, 89, 90]. The values that we extracted from such papers follows:

- $\epsilon_b = 0.05, 0.2$
- $\epsilon_n = 0.006, 0.0006$
- $\alpha = 0.5$
- $\beta = 0.005, 0.0005$
- $\lambda = 100, 300, 500$

[1]In [63], Fritzke says that, in case of a finite training set, a common measure is the number of cycles through all training patterns, also called *epochs*. Practically, an epoch is equivalent to a FACS-iteration.

- $a_{max} = 50, 88, 120$

We can see that, for α, a single value has been used in all of the examples reported. On the contrary, different values in different examples have been used for the other parameters. So, we have realized several tests for choosing their best combination; we fixed e_T = RMSE$_{15}$(128) and tried all of the possible configurations. In the end, we have verified that, for this problem, the best results were obtained by choosing $\epsilon_b = 0.2$, $\epsilon_n = 0.0006$, $\beta = 0.0005$, $\lambda = 500$, $a_{max} = 50$. We have also executed some tests with GNG-U, but, if using, for this problem, the same values of the parameters used by Fritzke , GNG-U performs worse than GNG. According to such considerations, we have completed our tests with GNG, changing the final RMSE and using the values determined above. The results obtained are summarized in Table 6.3. In Table 6.3 two series of values are reported: the former (labeled Ep_{min}) refers to the values obtained after the least number of epochs necessary for the achievement of e_T; the latter refers to the values obtained when the algorithm goes on until a maximum of 50 epochs, in order to better locate the codewords. N_{ep} is the number of epochs. As regards the test whose target is RMSE$_{15}$(4096), the algorithm is stopped as soon as e_T is reached, because of the high number (136) of epochs required. By comparing Table 6.3 with Table 6.2, we can clearly see that GNG ensures the desired target with a higher number of codewords. Further, when the e_T decreases, it also needs more epochs than FACS for ensuring the target.

e_T		Ep_{min}	Ep_{max}
RMSE$_{15}$(128)	N_{ep}	5	50
=	N_C	165	165
29.26	$\frac{N_C}{128}$	1.29	1.29
	RMSE	29.20	28.96
RMSE$_{15}$(512)	N_{ep}	19	50
=	N_C	624	624
22.39	$\frac{N_C}{512}$	1.22	1.22
	RMSE	22.14	21.88
RMSE$_{15}$(4096)	N_{ep}	136	-
=	N_C	4456	-
10.81	$\frac{N_C}{4096}$	1.09	-
	RMSE	10.78	-

Table 6.3: GNG performances

6.8.4 Comparison with FOSART

Now, we present a comparison with a technique belonging to the family of the Adaptive Resonance Theory (ART) algorithms proposed by Baraldi and Alpaydin in [48,69] and called Fully self-Organizing Simplified Adaptive Resonance Theory (FOSART). The authors themselves use it also for tasks of VQ. However, FOSART is not an algorithm designed exclusively for VQ, but it can also be used for problems of hidden data structure detection (perceptual grouping) and probability density functions estimation.

The parameters we have used for our tests are the ones used by the authors as default values, while the only quantity we have made change is the vigilance threshold ρ. The termination condition used for FOSART is the same as the one employed for ELBG with $\epsilon = 0.0001$ and the MSE as measure for the MQE. The range of values chosen for ρ is such that the number of codewords obtained is inside the range of values of N_C considered for the previous comparisons with ELBG and GNG. The results of the comparison are reported in Table 6.4. The first column represents the value of ρ FOSART was launched with. The final MSE obtained by FOSART was given to FACS as target; in the table, the number of codewords determined by FACS after 15 iterations is reported. As we can see, under the same MSE, FACS needs a number of codewords that is clearly less than the number of codewords used by FOSART.

FOSART				FACS
ρ	N_C	$N_{it(\text{FOSART})}$	MSE	N_C (15 it.)
0.0002	92	122	1109.7	65
0.0008	205	111	880.18	124
0.0010	323	84	735.42	194
0.0020	951	96	437.16	706
0.0050	2604	74	199.40	2423
0.0080	3816	59	131.16	3701

Table 6.4: FOSART comparison

6.8.5 Comparison with the Competitive Agglomeration Algorithm

Here, we report a comparison with the technique proposed by Frigui and Krishnapuram in [45]: the Competitive Agglomeration algorithm. FACS and the Competitive Agglomeration algorithm share the property that the number of the codewords (or *prototypes*, according to the nomenclature in [45]) has not to be specified.

Let us consider, for example, one of the data set used in [45]. It is reported in Fig. 6.10 together with the 4 codewords obtained by the Competitive Agglomeration algorithm after 10 iterations. Once the RMSE previously obtained has been fixed as the target, FACS, in only 6 iterations, finds 4 codewords located as in Fig. 6.11 and the RMSE obtained is practically the same (about 99.8%) as the one of the Competitive Agglomeration Algorithm.

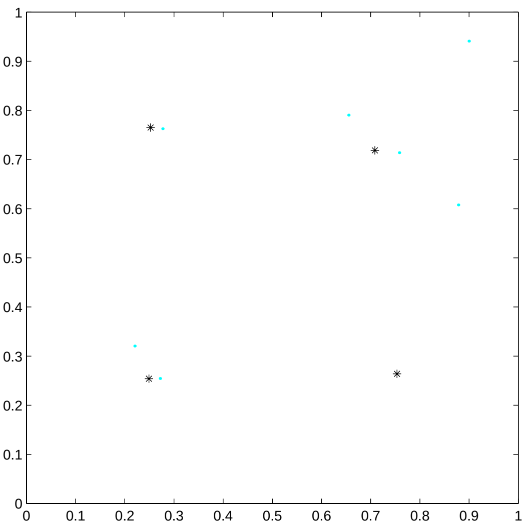


Figure 6.10: Prototypes of the Competitive Agglomeration Algorithm after 10 iterations

6.8.6 Classification

The subject of this sub-section is a comparison between FACS and the Growing Cell Structures (GCS) algorithm, another technique proposed by Fritzke in [63] and used, in the same paper, also for a problem of supervised classification. Also in this case, the task of finding the right number of codewords is left to the algorithm.

The test mentioned above concerns the problem of the two spirals: it consists of 194 two-dimensional vectors lying on two interlocked spirals, which are the classes to be distinguished in this case.

The whole procedure of the classification is constituted by several phases, one of which (the clustering phase) consists in the execution of FACS. We have executed our tests operating in two distinct modalities. In the former,

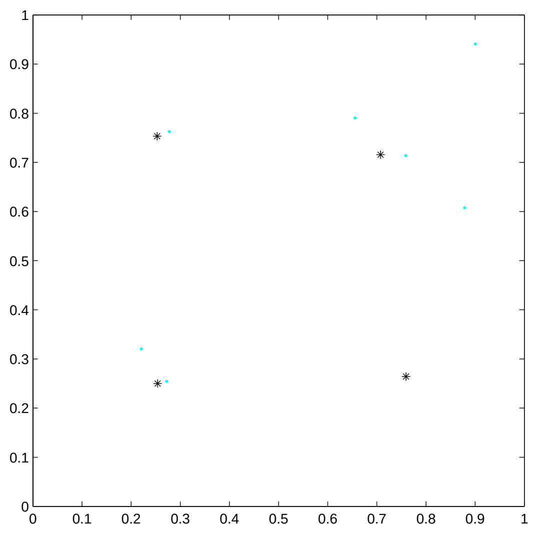


Figure 6.11: Codewords of FACS after 6 iterations

the labels (class 0 or class 1) of the points constituting the spirals are used during the clustering phase; in the latter, the clustering phase occurs without using the labels that are employed in the following phase, i.e. the labeling of the codewords.

Mode 1

In this case, the input values are given, together with the related outputs, to the clustering system. The input is constituted by 194 bi-dimensional vectors representing the two spirals, while the output is the related membership class (0 or 1).

The procedure we have followed to perform the classification can be summarized in the three following points.

1. **Creation of the learning patterns to train FACS.** They are constituted by the tri-dimensional vectors obtained by joining each of the 194 two-dimensional vectors with the related output as the third component.

2. **Clustering Phase.** FACS is launched giving it the tri-dimensional vectors constructed at the previous point as the input data set. Later, we will see the details of this point.

3. **Codewords labeling.** In order for the classification to happen correctly, the execution of the opposite operation of the one at point 1

is necessary. Therefore, in each tri-dimensional codeword, the components related to the input have to be separated from the ones related to the output. This, as we have done for the learning patterns, is used to label the related bi-dimensional codeword. The codeword will be identified as representing class 0 or 1 whether respectively more bi-dimensional patterns belonging to class 0 or 1 are present inside the cell in question (majority voting, [91]).

4. **Classification.** Each (bi-dimensional) input pattern is compared with all of the labeled bi-dimensional codewords. Once the codeword with the least distance from the pattern in question has been found, its label is read and the pattern is considered as belonging to that class.

The following specifications are necessary so that the execution of FACS (point 2) occurs correctly.

- **Pre-processing of the data.** When the components of the vectors constituting the input patterns all have very different statistical distribution (as regards both the order of magnitude and the shape of the distribution) some problems can arise. In fact, it is possible that the components with higher dynamics are approximated with high precision, while the ones with lower dynamics are approximated with poor precision. In order to avoid such a situation, the tri-dimensional vectors constituting the input data set for FACS, have been pre-processed by normalizing each component with respect to standard deviations and, for all of the subsequent operations, the vectors so obtained have been employed;

- **The distortion measure adopted.** We tried to force FACS to generate a codebook with cells of tri-dimensional vectors as *homogeneous* as possible . With the term *homogeneous* we mean that a cell is constituted by patterns belonging to the same class, i.e. the third component of its vectors is the same for all of them. To realize this condition, we employed the WSE (eq. (2.15)) as the distortion measure and we assigned a very high weight to the output component with respect to the one assigned to the input components. In particular, we chose $w_1 = w_2 = 1$ and $w_3 = 100$ (the same result was obtained also for $w_3 = 10$ and, obviously, with $w_3 = 1000$). So, the clustering algorithm is forced to divide the patterns by favouring first their membership class and then also the neighbourhood in the bi-dimensional space. We have verified that, for the problem in question and with the value of the weights reported above, already with a value of e_T that generates only two codewords, the related cells are entirely homogeneous. Of course, this does not imply that two codewords are enough for a correct classification of the data.

The evaluation of the performances occurs in terms of both classification error and number of iterations executed.

- **Classification error.** Figs. 6.12 and 6.13 graphically report the results found when the values $e_T = 0.01$ and $e_T = 0.001$ respectively have been used. In both cases, FACS runs for 15 iterations. In the figures mentioned, points represent learning patterns and circles represent codewords. The decision regions (the black and white ones in the figures) determined by the codewords are reported, too. For $e_T = 0.01$, FACS found a codebook with $N_C = 74$, while, for $e_T = 0.001$, a codebook with $N_C = 144$ was found. In the first case, there is only one error in the classification of the learning patterns; in the second case no errors are committed and, as we can see from the figure, the decision-regions are well defined also when we are not in proximity to learning patterns. Even if FACS runs for 200 iterations, it obtains the same number of codewords found after 15 iterations.

  Afterwards, we have executed the testing of the results obtained by using the same test set employed by Fritzke, constituted by 576 points. He compares this result with the one obtained by Baum and Lang [92] that, for their best model, report an average of 29 errors on the test set. Fritzke, with 145 codewords, obtains zero errors on the same test set; FACS too achieves this result with the 144 codewords of Fig. 6.13.

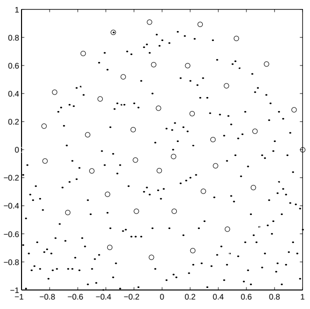


Figure 6.12: Two spirals: supervised classification. $e_T = 0.01$; $N_C = 74$

- **Number of iterations.** In [63] Fritzke underlines that, for every learning method, an important practical aspect is the number of pattern

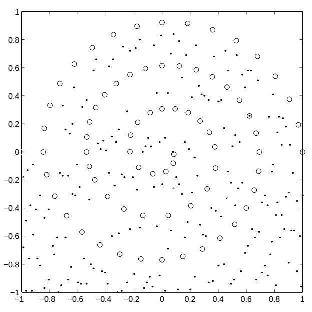


Figure 6.13: Two spirals: supervised classification. $e_T = 0.001$; $N_C = 144$

presentations necessary to achieve a satisfying performance. Fritzke says that GCS needs 180 epochs to achieve the above mentioned result and reports comparisons with some earlier methods: Backpropagation [93] (20000 epochs), Cross Entropy BP [93] (10000 epochs), Cascade-Correlation [94] (1700 epochs) and he says that the number of epochs required by GCS is about one or two orders of magnitude less than the other techniques reported.

We can see that FACS needs about 15 iterations to obtain the same result as the GCS, i.e. a twelfth of the epochs required by the GCS. However, after a careful analysis of both FACS and GCS, we have decided that a comparison performed in these terms is not very precise. In fact, the complexity of the epochs (iterations) can be different for each of them. From the analysis effected with several profiling tools we have seen that, in similar algorithms, almost the whole computation time is spent comparing the vectors of the input data set with the vectors of the codebook (distance calculation). So, we can identify the complexity of an epoch (iteration) as the number of comparisons executed inside it. But, due to the incremental nature of such algorithms, the number of codewords is not the same for all the iterations, therefore the complexity of each iteration is different, too.

As regards FACS, for the example reported in the previous subsection, we counted automatically the comparisons effected and, for $e_T = 0.001$ (that produced the codebook with 144 elements), we found a value of about 0.6 milion for the 15 iterations executed.

For GCS, we effected the calculation starting from the values given by Fritzke. It begins with 3 neurons (that are equivalent to our codewords) and, every 240 pattern-presentations, inserts a new neuron until the final number of 145 cells is reached and for a total number of 180 epochs.

We estimated that this is equivalent to the execution of about 2.6 milion of comparisons, that is about 4.3 times the number of comparisons effected by FACS.

Mode 2

The test on the two spirals problem has been treated also working in another mode. From the practical point of view, the steps to be executed are the same as the previous ones but, now, the clustering phase occurs using only the part of the patterns related to the input (i.e. the original bi-dimensional patterns) without adding the membership class as the third component. Besides, we use the SE (eq. (2.14)) as the measure for the distortion.

Giving to FACS $e_T = 0.01$, it has obtained $N_C = 69$ codewords, producing 44 classification errors for the learning patterns. Using, $e_T = 0.001$, it has found $N_C = 143$ codewords and 0 errors have been obtained both on the learning and in the testing set. The results related to the last example are reported in Fig. 6.14; we can see that this is very similar to Fig. 6.13. Also in this test, FACS was stopped after 15 iterations and the same considerations about the computational complexity of the iterations made for the previous mode can be repeated here.

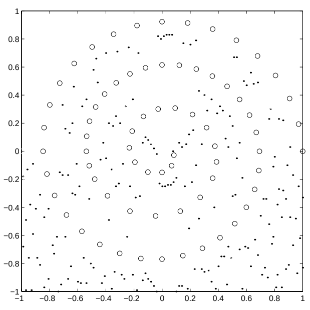


Figure 6.14: Two spirals: unsupervised classification. $e_T = 0.001$; $N_C = 143$

6.9 Conclusions

In this chapter FACS has been presented, a new algorithm for clustering and vector quantization that is able to autonomously find the number of codewords once the desired quantization error is specified. The technique uses some concepts previously developed for ELBG which works with a prefixed number of codewords and rearranges them smartly to escape from the local minima of the error function. Comparative studies regarding FACS have shown that it is able to find good results in very few iterations. For complex clustering applications only 15 iterations are sufficient. In comparison to previous similar works a significative improvement in the running time has been obtained.

Further studies will be made regarding the use of different distortion measures to obtain detection of elipsoidal and linear clusters, and planar range segmentation [14]. Other studies will regard the possibility of dealing with input patterns with variable cardinality.

Chapter 7

The MULTISOFT Machine

7.1 Introduction

In this chapter, the MULTISOFT machine is described [7]. It is a cluster of 32 PCs, running the LINUX operating system and using the Parallel Virtual Machine (PVM) software[1] [95, 96] for inter-process communications.

The policy adopted for the management of the MULTISOFT machine permits a very fast increase of the number of the Processing Elements (PEs), that is the number of PCs. It is based on a completely centralized administration of the system, but particular care was taken in order to minimize the network bandwidth required in normal operative conditions. Several similar machines exist in the world [97][2]. Many of them are also based on centralized administration, and, in many cases, a quota of the interconnection network bandwidth is needed for the sharing of the file system. This aspect is very important above all for systems like the MULTISOFT Machine, where the total bandwidth is very low. Generally, the interconnection network in multicomputers is the bottleneck of all well-implemented distributed algorithms, so its under-utilization directly affects the maximum value of the achievable speedup.

7.2 Hardware organization

The MULTISOFT machine (see Fig. 7.1) is a cluster of PCs with the LINUX OS. It consists of 32 Processing Elements (PEs or hosts) and a server (duplicated for redundancy) acting also as a gateway towards the external LAN.

[1]It is available together with XPVM for free at http://www.netlib.org/pvm

[2]A 1000-pentium based machine employed for applications regarding genetic programming is described at http://www.genetic-programming.com/machine1000.html

All of the 32 PEs are mono-processor machines, 7 are based on Pentium-II 233MHz (we call them p233) and 25 on Celeron 366MHz (c366). All of them are endowed with 32 MB of RAM, one hard disk (ranging from 3.2 to 4.3 GBytes) and one Fast-Ethernet network card. One server (Server01) is a bi-processor computer with two Pentium-II 300MHz and two network cards and the other (Server02) is a mono-processor Pentium II 350MHz.

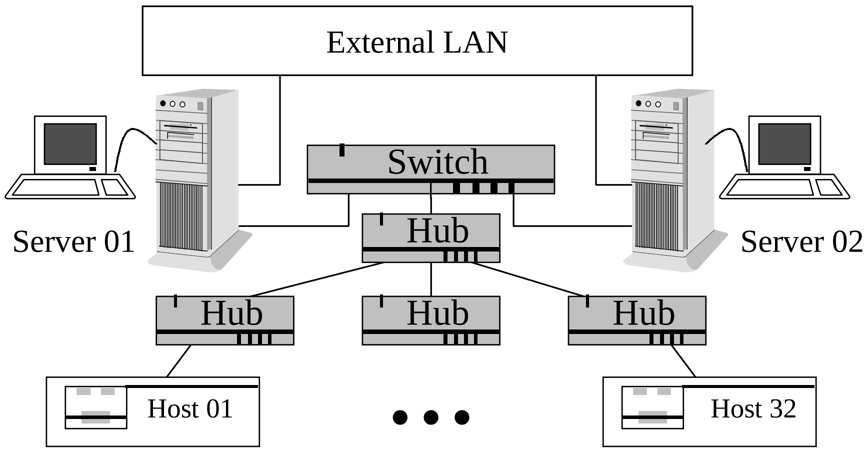


Figure 7.1: The structural of the MULTISOFT machine

The PEs are interconnected by a dedicated Fast Ethernet network and all of them share the same bus. Physical connections have been realized by means of four 12-port hubs distributed between two levels as we show in Fig. 7.1. However, the electrical topology is such that all of the 32 hosts are placed on the same bus. The highest level hub is connected to a switch, where the servers are directly connected, too.

Inter-process communication is realized with the Parallel Virtual Machine (PVM) software. It allows the user to see several computers as a large and single virtual machine and every process (task, according to the PVM nomenclature) has a unique identifier (task identifier, tid) inside the virtual machine. The PVM uses the transport functionality provided by the TCP/(UDP/)IP that are directly implemented inside the kernel of LINUX. Together with the PVM, we employ the graphic tool named XPVM; it allows us to visualize the diagram of the activity of the tasks versus time. It is very useful during the debugging and the testing of parallel algorithms. Further, it allows distributed algorithms to be described well, as will be shown in the following chapters.

7.3 Management

For the management of a system with a high number of hosts, all sharing the same configuration, we thought it was suitable to pursue a centralized policy allowing the administrator to quickly modify the configuration of the whole system without having to operate on the single machine. Besides, when the system expands, we want to be able to quickly set up the new machines.

The first solution we considered was the realization of a system similar to the one described in [97]. All of the hosts were disk-less machines and remote-boot and Network File System (NFS) were employed. In this way, each host shared most of its file-system (physically stored on the server) with all of the other hosts. Files that cannot be shared were stored on the server, too, but were kept in distinct directories, one for each host. These are the files containing information related to the identification and the status of the single host, its peripherals and the processes running on it. Therefore, they are different for each of them and, for this reason, they cannot be shared. Such a policy allows a quick update of the system because each modification effected on the shared portion of the file-system of a host, automatically and immediately, regards all of the other hosts. On the other hand, changes regarding the unshared portions are much less frequent. However, when such a centralized solution is adopted, we have to consider two main problems. The first concerns the access to a NFS, much slower than a local file-system. The second arises from the increase in the network traffic, with the consequent significant reduction of the bandwidth available to inter-process communication.

On the basis of these considerations, we found two, apparently, opposite requirements: the centralization of the information (in order to simplify the management of the system) and the maximization of the performances by saving bandwidth, so that it is completely at the disposal of inter-process communication.

The solution we chose is a hybrid one between centralized and local management. Under normal conditions of working, all of the hosts use a file-system stored on the local disk. Nevertheless, it is a copy of a file-system that is physically stored on the server. It is the same for everyone and it is downloaded by the hosts in an almost completely automatic way. Such an operation of downloading is necessary only when modifications or failures in the hosts occur. So, we need to take care of the configuration of only two machines: the server and a host. In fact, once a generic host has been configured (upgraded), its file-system is transferred to the server and from there it is, automatically, downloaded by all of the other hosts.

Before describing in greater detail the policy we adopted for the management of the system, we wish to say that, according to our terminology, each host can operate in one of the three following modalities:

- **Managing.** It is used for the first installation or if a failure occurs.
- **Service.** It is used to update the hosts.
- **Operative.** It is the normal modality of working.

A host enters a certain modality when it boots. When it is necessary to change the modality, its configuration is opportunely modified so that, after the reboot, it enters the desired modality. Each host can operate in any modality with no console (keyboard, monitor, mouse). Only the server has a console.

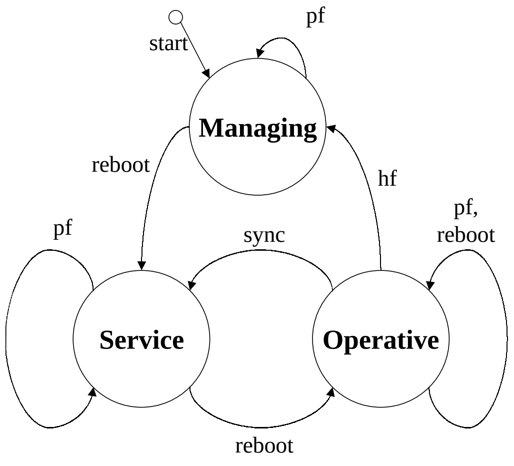


Figure 7.2: The FSM describing the OS organization of the MULTISOFT machine

The three modalities and the transitions from one modality to another, can be represented by means of a Finite State Machine (FSM), as we report in Fig. 7.2. The signals that make the host change (or remain in) its state are the following:

- start: it is used to begin the first installation of the host. It indicates that a new PE must be physically and logically added to the system, or that an existing PE has to be reconfigured *ex novo* after a failure.
- power failure (pf);
- reboot: it represents the normal reboot of the host;

- sync: it is used to force the host to download from the server the most recent version of the file-system related to the operative modality;
- host failure (hf): it indicates that the host is not working correctly.

Now we will detail the operations effected in each modality and how the transition from one modality to another occurs.

- **Operative modality.** This is the modality where the host, normally, operates. It boots and works by using the local operative file-system (that is the same for all of the hosts). Only the information related to the host identification on the network (its name and its IP address) differs for each of them. However, these data are not physically stored on the host, but are all on the server which communicates them to the host when it boots by means of a protocol such as bootp or dhcp. Even if the host works by using a local file-system, the auditing of the user accounts is effected by the server through the NIS protocol. Besides, the home directories of the users are shared by means of the NFS. However, users have also a personal local directory on each host, so that they can choose to use the shared, the local directory or both of them. If the host is in this state and a reboot or a power failure occur, it reboots and remains in the same state. Instead, if the signal of sync is raised, the host opportunely modifies its configuration and reboots in the service modality.
- **Service modality.** This is a transitory modality. It is used by the hosts to download the file-system that will be used in the operative modality from the server. In this state, the kernel mounts a reduced file-system, identical for all of the hosts, previously downloaded from the server, exactly as it happens for the operative file-system. Also in this case, the information identifying that host in the network are assigned by the server during the boot phase. The service file-system contains only the files that are necessary to the host so that it can download the operative file-system from the server and store it on the local disk. If no power failures or other generic failures occur, the host modifies its configuration and reboots in the operative modality. All of these operations are executed automatically. If a power failure occurs, the host reboots in the service modality, while, if a generic failure occurs, it is rebooted in the managing modality by inserting the proper floppy disk.
- **Managing modality.** When a host operates in this modality, it is able to work without using the local disk, as if it was a disk-less machine.

All of the files necessary to effect the remote-booting are stored on a bootable floppy disk. After the kernel has been loaded, the host mounts the file-system related to this modality by NFS. At this point, all of the operations relating to the initialization of the local disk (partitioning and formatting) necessary so that this can hold the service and the operative file-system, are executed. Therefore, the service file-system is downloaded from the server and the configuration is modified so that the host can reboot in the service modality. The reboot gets the hosts into the service modality, from where, if no failures occur, it automatically enters the operative modality. If a power failure or a generic failure occur the host reboots in the managing modality.

7.4 Updating the system

In the previous subsection we described the modalities where each host of the system can operate. Now we will describe how its update is effected. This is necessary, for example, when a program is upgraded or installed *ex novo*. The modifications are made on any host working in the operative modality. Afterwards, its operative file-system is compressed and transferred to the server. Now, it is enough to raise the "sync" signal for all of the hosts and these will reboot in the service modality. Therefore, as we previously described, they download the most recent version of the operative file-system from the server and, when they finish, reboot in the operative modality. So, we can easily transfer the changes we made on a single machine to all of the system and in a completely automatic way . We must remember that this would be possible if the hosts mount their whole file-system by NFS but, as we said, this would make the system work more slowly and would consume transmission bandwidth. Instead, our method allows us to automatically update the system without overloading the network when it works in the normal operative modality.

Chapter 8

Parallel Implementation of LBG and ELBG

8.1 Introduction

In this chapter we present a preliminary study regarding the parallel implementation of techniques for unsupervised learning on the MULTISOFT machine (described in chapter 7) [6, 7]. Two algorithms have been considered: LBG (presented in chapter 3) and ELBG (presented in chapter 4). The methodology adopted behaves poorly in the case of simple clustering problems. The lack of a true broadcasting function in our communication system makes everything worse. However, when the number of required computations grows, making the clustering task more complex, the results are encouraging because the speedup significantly increases. This means that commodity supercomputers are very suitable for very complex unsupervised problems. The chapter is organized as follows: in Section 8.2 the methodology adopted for the parallelization of LBG and ELBG is described; in Section 8.3 the results are presented; lastly, 8.4 reports the author's conclusions.

8.2 Parallelization of LBG and ELBG

In chapter 4, we saw that the overhead introduced on the traditional LBG by the execution of the ELBG block is quite low. Besides, the profiling executed on the software written to implement LBG allowed us to see that almost all of the calculation time is spent to determine the Voronoi partition (point 2 of the LBG algorithm). More precisely, this time is spent to calculate the distance between each input pattern and all of the codewords, in order to choose the nearest one. Our work focused on the parallelization of this operation by trying to share the load among all of the hosts. Now, we will describe our parallel implementation of LBG and, after, the parallel version of ELBG, that is identical to LBG only with the addition of the ELBG block. The parallel versions of LBG and ELBG will be called PARLBG and PARELBG, respectively, in the remainder of the thesis.

8.2.1 PARLBG

As we previously said, during the calculation of the Voronoi partition, each input pattern is assigned to the nearest codeword to it. As this is done by comparing each input vector with all of the codewords, it means that the distance (Euclidean, in this case) between two k-dimensional vectors is calculated $N_P \times N_C$ times. If we have N processors at our disposal, we can launch, on each processor, a process (task, according to the PVM nomenclature) that executes $\frac{N_P \times N_C}{N}$ of distance calculations. In practice, each task keeps in the memory a different portion of the input patterns and the whole codebook. It has to find the nearest codeword only for the portion of input patterns that were assigned to it.

Given N equal hosts, we adopted a master-slave policy. At each iteration, the master calculates, together with the slaves, the Voronoi partition, collects the results and, finally, calculates the new codebook alone. At the beginning of the new iteration, the new codebook is distributed, in broadcast, to all of the slaves.

Fig. 8.1 is a screen-shot taken from XPVM. It reports a part of the diagram of the tasks activity versus the time. It refers to a run of PARLBG on 4 hosts, with an input data set of 16384 16-dimensional vectors and a codebook of 128 elements. The input data set was obtained from the image of Lena [86] of 512×512 pixels at 256 grey levels. It was divided into blocks of 4×4 pixels, and, in this way, 16384 16-dimensional vectors were generated. The figure illustrates the periods of activity and inactivity of the tasks, the exchange of the messages and the overhead introduced by their transmission. XPVM does not make the graphic tracing of the broadcast messages[1], that in our algorithm, are the messages relating to the transmission of the codebook from the master to the slaves. The scale of times was divided into parts labeled as (a), (b), (c) and (d); now, we will describe the operations executed during each of them.

(a) Preliminary phase. The master spawns the slaves and waits until all of them start correctly.

[1]PVM provides a function of sending data in broadcast to all the tasks; however, this is not a real broadcast function because it is implemented by sending the same copy of the data to all the tasks, one at a time.

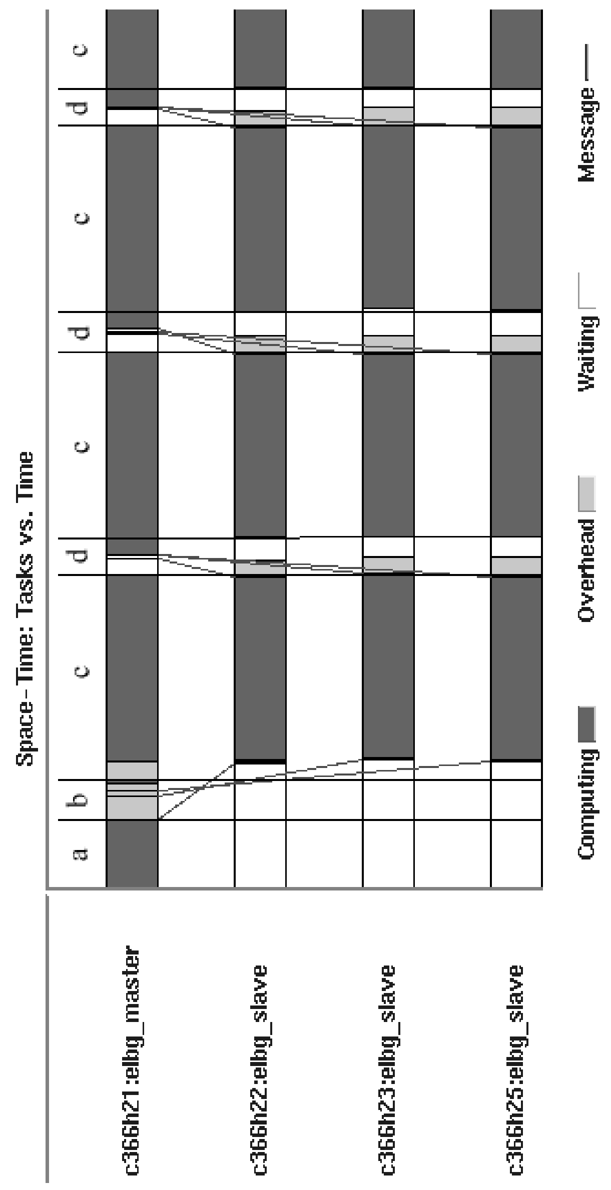


Figure 8.1: Task vs. Time diagram for PARLBG with $k = 16$, $N_P = 16384$, $N_C = 128$, $N = 4$. The scale of times is such that an iteration is about 0.75 s.

(b) Distribution of the input data set. In this phase, the master distributes (just once), to each slave, the portion of input patterns for which it has to calculate the nearest codeword.

(c) Voronoi partition calculation. This is the heart of the parallel algo-

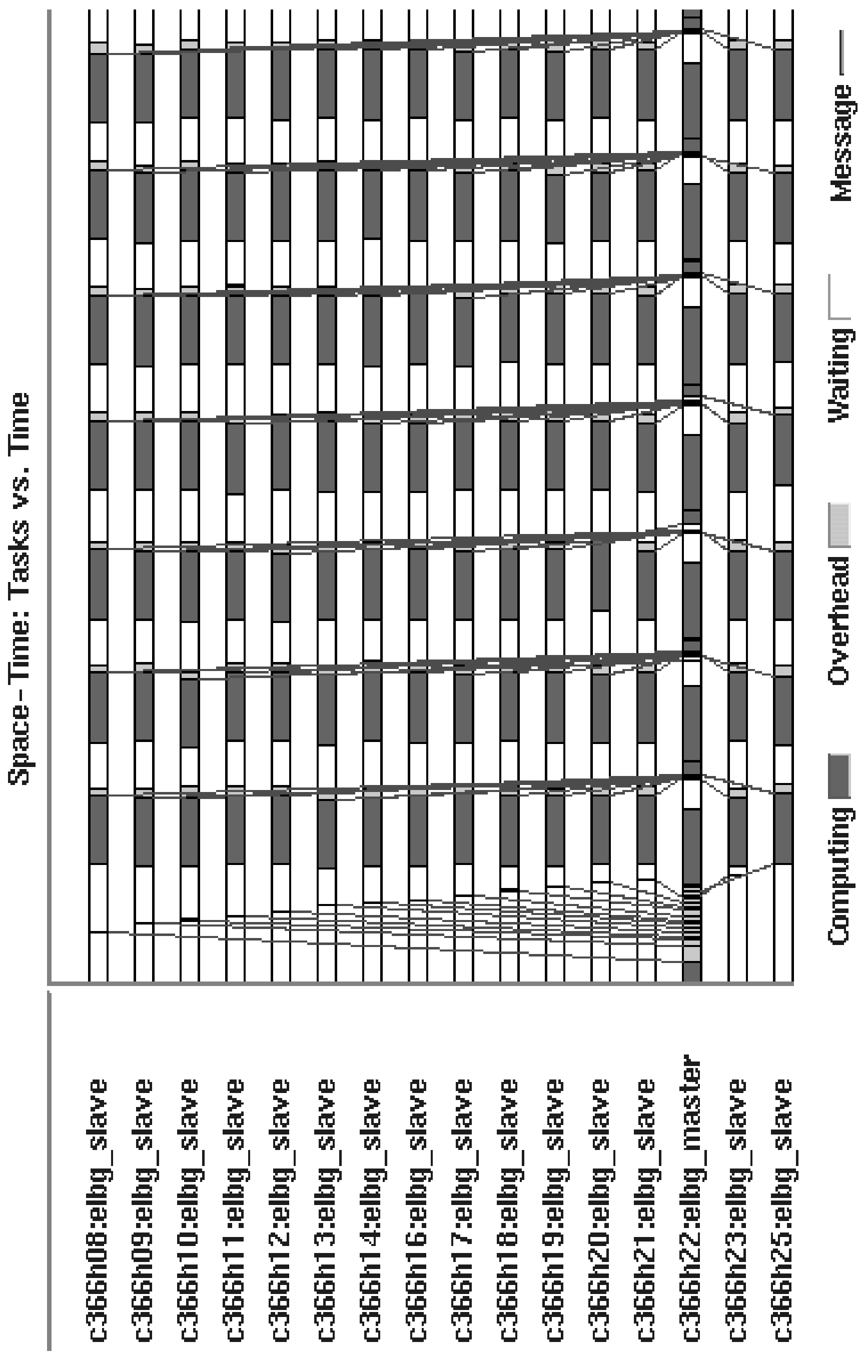


Figure 8.2: Task vs. Time diagram for PARLBG with $k = 16$, $N_P = 16384$, $N_C = 256$, $N = 16$. The scale of times is such that an iteration is about 0.60 s.

rithm. The first operation in this phase is the distribution, in broadcast, of the codebook from the master to all the slaves. Even if we cannot see the graphic tracing of the broadcast message, we can see that, for the considered example, the transmission of the codebook introduces a negligible delay with respect to the whole phase. In fact, by carefully looking at all segments labeled as (c), we see that all of the slaves, practically, start working at the beginning of the phase. This is not true for the first iteration (the one after segment (b)), but, from the same picture, we can see that such a delay is the consequence of the messages transmitted during phase (b) still occupying the network. Phase (c) ends when the master completes the calculation of its quota of the Voronoi partition.

(d) Collection of the results from the slaves and calculation of the new codebook. This is the critical part for the parallelization; in fact, during its execution, only the master works, while the slaves, after they have sent their results, wait for the new codebook from the master. Therefore, it is opportune to try to minimize its duration. We effected several tests of the algorithm when k, N_P, N_C and N change and we analyzed the results, both the graphic ones from XPVM, and the numerical ones from the profiling of the master. We concluded that, for the implementation of PARLBG in our system, it is better to minimize the information transmitted on the network, but, in any case, not loading the master with the calculation of the Euclidean distance. This is calculated by the master only during phase (c) when it determines its portion of the Voronoi partition. According to these considerations, the information transmitted from a slave to the master is constituted, for each pattern, by the index of the nearest codeword and by the QE related to it. Each slave sends these results when it completes the calculation of its portion of the Voronoi partition. When all of these values have been collected by the master, it is able to calculate the total distortion and the new codebook. The master cannot calculate the new codebook until all of the data are received. In order to minimize its periods of inactivity, it operates as follows: it checks whether the results from any slave have been received and executes all of the operations it can execute with those data; after, it checks whether there are other results, and so on until it receives all of them. From the profiling of the master, effected for $N = 1, 2, 4, 8, 16$, we saw that the time it employs to calculate the new codebook, after all the results have been collected, is less than 1%; so, it can be considered negligible.

Fig. 8.2 refers to the same input data set employed for the example of Fig. 8.1 and, in this case, we have $N_C = 256$ and $N = 16$. We can see that the periods of inactivity of the tasks increase when N increases, too. This is due to the greater quantity of data to be transmitted over the network. Therefore, when N reaches a certain value, the adding of other processors does not produce any benefit, i.e. the speed-up saturates.

8.2.2 PARELBG

As we previously said, our implementation of PARELBG is substantially identical to the implementation of PARLBG with the addition of the ELBG block. It is executed, in serial mode, only by the master.

Fig. 8.3 reports a part of the diagram of the activity of the tasks versus the time, for the same problem we considered in Fig. 8.1 for PARLBG. The meaning of segments (a), (b), (c) and (d) is the same as in Fig. 8.1 with the only difference that, in this case, inside phase (d), before the new codebook calculation, the master also executes the ELBG block. In order to visualize a greater number of iterations, in Fig. 8.3, we chose a different scale for the axis of the times with respect to Fig. 8.1. However, we can take as reference the time of "Computing" of the slaves at each iteration; it is the same both in PARLBG and in PARELBG.

The first difference we can notice between Figs. 8.1 and 8.3 is that, in the latter, the periods of inactivity of the slaves are longer than in the former. This has to be attributed to the execution of the ELBG block by the master; such an operation makes segments (d) longer than in PARLBG. We verified that the periods of inactivity are reduced when the complexity of the problem increases. In the cases we analyzed, an increase in the complexity consists in the increase of N_C, N_P being fixed. Besides, as we can see in Fig. 8.3, the length of segments (d) is not fixed because of the ELBG block, whose computing time is not deterministic as in the calculation of the Voronoi partition.

Another difference between the diagrams, that we think could be imputable to the pair PVM-TCP/IP, is related to segments (c). Both in PARLBG and in PARELBG, the master transmits the new codebook to the slaves at the beginning of (c) and, as the dimensions of the codebook are the same in both of the cases, the quantity of data to send over the network is the same. However, the time required for the transmission is negligible in Fig. 8.1, while it is not in Fig. 8.3. We repeated the runs several times, but the result did not change. We think this could derive from different behaviour of the pair PVM-TCP/IP in the two situations we examined. As a consequence of this, in PARELBG, the period of inactivity of the master at the beginning of segments (d) is longer than in PARLBG.

		PARLBG		PARELBG	
N_C	N	*T it. (s)*	S	*T it. (s)*	S
32	1	0.62	1.00	0.79	1.00
	2	0.48	1.30	0.64	1.23
	4	0.23	2.73	0.40	1.97
	8	0.16	3.89	0.33	2.40
	16	0.13	4.60	0.32	2.49
64	1	1.17	1.00	1.41	1.00
	2	0.76	1.54	1.18	1.19
	4	0.45	2.60	0.87	1.61
	8	0.24	4.89	0.75	1.88
	16	0.19	6.26	0.73	1.93
128	1	2.28	1.00	2.60	1.00
	2	1.31	1.73	1.85	1.40
	4	0.75	3.05	1.27	2.04
	8	0.49	4.69	1.00	2.61
	16	0.28	8.17	0.91	2.86
256	1	4.51	1.00	5.00	1.00
	2	2.44	1.85	3.17	1.58
	4	1.31	3.44	2.02	2.48
	8	0.77	5.83	1.48	3.39
	16	0.60	7.51	1.25	4.00
512	1	9.30	1.00	9.90	1.00
	2	4.77	1.95	5.77	1.72
	4	2.49	3.73	3.50	2.83
	8	1.37	6.79	2.37	4.18
	16	1.03	9.02	1.86	5.32
1024	1	18.56	1.00	20.47	1.00
	2	9.39	1.98	11.10	1.84
	4	4.77	3.89	6.94	2.95
	8	2.59	7.17	4.41	4.65
	16	1.65	11.24	3.22	6.35

Table 8.1: Speed up (S) for PARLBG and PARELBG

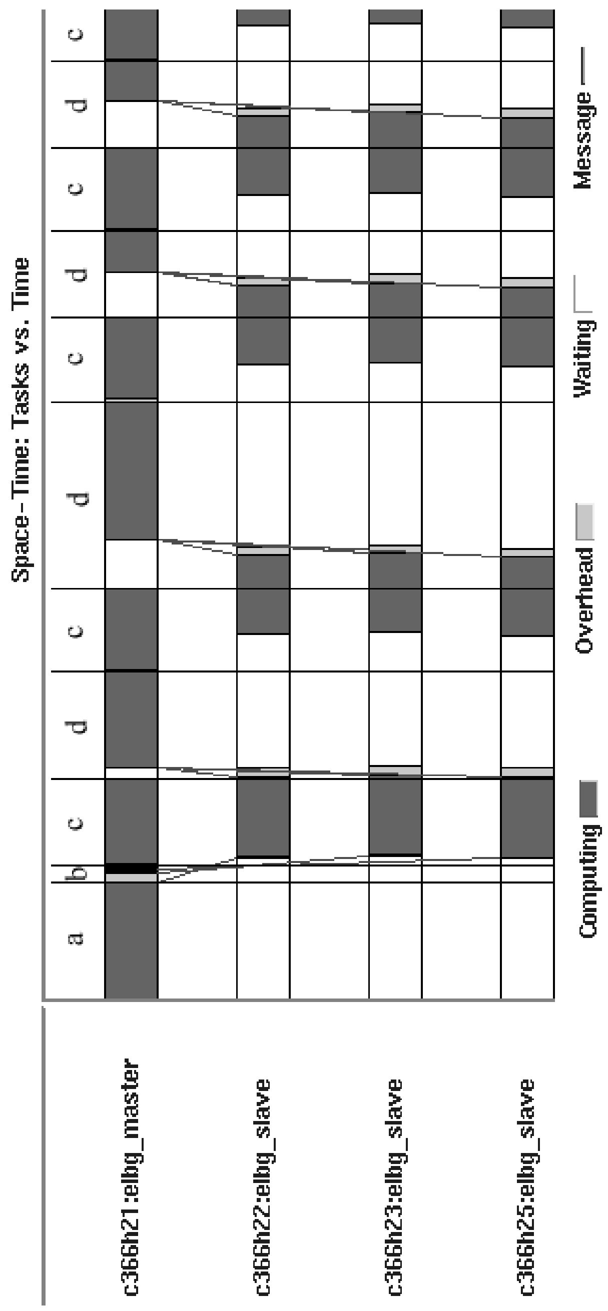


Figure 8.3: Task vs. Time diagram for PARELBG with $k = 16$, $N_P = 16384$, $N_C = 128$, $N = 4$. The scale of times is such that an iteration is is about 1.27 s.

8.3 Results

In this section we report the performances of PARLBG and PARELBG in terms of speed-ups (S). Performances, in terms of final MQE and number of iterations required for the convergence, are identical to the ones obtained by the corresponding serial versions. On this subject, several comparisons with other existing techniques [38, 39, 53, 59], both hard and fuzzy, both k-means and competitive learning, are reported in chapter 4 and in [1–3]. In such comparisons, ELBG obtained results better than or equal to all of the other considered techniques, as regards both the final MQE and the number of required iterations. Besides, the difference in favour of ELBG increases when the complexity of the clustering problem increases, too and the final MQE ELBG obtains is, practically, independent of the initial conditions.

Before reporting the numeric values related to S, we think that some considerations about the validity of the results obtained regarding the times of calculation are worthwhile. LBG is a deterministic algorithm, i.e., after the initial codebook has been fixed, it always develops in the same way. The same deterministic behaviour has to be followed both by the serial and by the parallel versions. However, the utilization of a Fast Ethernet-based network, whose management is not deterministic, gave us slightly variable results. Regarding ELBG, we must also consider the non-deterministic behaviour of the ELBG block. So, in order to make allowance for these factors, all of the results we present are the mean value of 20 runs. In Tab. 8.1 we report the performances of PARLBG and PARELBG in terms of S with respect to the time required per iteration; the input data set is the same we adopted in the previous section. Some considerations regarding these results follow.

- We can see that, for all of the examined cases, PARLBG presents a higher value of S than PARELBG. This is obvious because of the serial execution of the ELBG block only by the master, while the time of activity for the slaves is the same as PARLBG.

- The number of processors (N) being equal, generally, S increases when the complexity of the problem increases, too. In this case, the complexity is determined by the dimensions of the codebook (N_C) because we fixed the input data set. Such a trend is justified by the decrease in the time of inactivity of the slaves. However, there are some cases where this trend is inverted. For example, let us look at PARLBG when we change $N_C = 32$ into $N_C = 64$ for $N = 4$, $N_C = 64$ into $N_C = 128$ for $N = 8$, $N_C = 128$ into $N_C = 256$ for $N = 16$. We saw that such behaviour derives from the increase of the overhead in the transmission of the codebook from the master to the slaves.

- N_C being fixed, when N increases, S saturates. This occurs because the increase of N prolongs the periods of inactivity of the slaves and, in practice, their contribution to the algorithm no longer increases. Both for PARLBG and PARELBG, when N_C increases, saturation occurs for higher values of N. Also in this case, it is a good trend because both of the parallel algorithms we considered perform better with high complexity problems.

8.4 Conclusions and future developments

The work presented in this chapter is a preliminary study regarding the parallelization of clustering algorithms on the MULTISOFT machine, a commodity supercomputer. In particular, we analyzed two techniques for VQ (LBG and ELBG) and derived a parallel implementation for each of them (PARLBG and PARELBG, respectively). We could see that some points, such as the calculation of the Voronoi partition, are easy to implement on the system we described, others can be improved. Regarding PARELBG, we are thinking of developing a new version of the ELBG block that could be executed in parallel by all of the hosts in order to increase the speed up of the whole algorithm. Another improvement, whose benefits would interest both PARLBG and PARELBG, can be introduced in the inter-process communication by the utilization of a real broadcast function. In fact, in the current implementation of the two algorithms, the broadcast transmission of the codebook from the master to the slaves is effected by PVM by simply sending the same copy of the data, one at a time. With this policy of broadcasting, the speed-up tends to decrease when the number of PEs increases a lot. However, for the number of PEs we considered in this work, such behaviour does not take place. With a function of real broadcast (as the one provided by the UDP on a Fast Ethernet network), the master could transmit the codebook on the network, just once per iteration, and all of the slaves could read it at the same time making the speed-up higher than at present. The last improvement regards the possibility to use efficiently a distributed system whose PEs are based on processors of different speed and/or types, as the MULTISOFT machine is.

Chapter 9

PAUL: A Parallel Algorithm for Unsupervised Learning

9.1 Introduction

The parallel implementation of an efficient serial clustering algorithm is, often, not very efficient because of the intrinsically serial nature of some of the operations executed. This is a strong limitation to the achievable speed up, independent of the available systems and the technique adopted for the parallelization.

The fundamental idea from which this work arises is to perform the parallelization of an algorithm by substituting intrinsically serial operations with efficiently parallelizzable versions that try to approximate the original ones as well as possible. On the one hand, we expect the deterioration of the final quantization error; on the other, we expect an improvement of the speed up with respect to the value we would get by parallelly implementing the original algorithm leaving all of the computational load to a single task. Sometimes, in the remainder of the chapter, we will use the terms *performance* when referring to the result obtained by an algorithm in terms of final error and *efficiency* when referring to its computational complexity.

The work presented in this chapter (and in [8]), implemented on the MULTISOFT machine (chapter 7), has been developed following the philosophy of compromise between performance and efficiency. It is an UL algorithm whose ideal field of application is constituted by very complex problems of unsupervised learning with a lot of both patterns and codewords.

The serial UL algorithm we considered is ELBG (chapter 4). By means of many examples, it has been demonstrated that ELBG achieves good performances, practically, independently of the initial conditions. Besides, the employment of sub-optimum techniques and the particular implementation (chapter 5) made it efficient. However, it belongs to that family of algorithms that cannot be efficiently implemented in parallel because of the serial nature of some operations it executes. In this context,it is important to consider the preliminary study carried out in chapter 8, where a non-approximated parallel implementation of both LBG and ELBG on the MULTISOFT machine, called PARLBG and PARELBG respectively, has been presented.

The analysis of the results obtained has been very useful for understanding how to modify ELBG in order to develop a new parallel algorithm representing a compromise between performance and efficiency according to the philosophy described above. The technique designed and implemented, constituting the subject of this chapter, has been named *Parallel Algorithm for Unsupervised Learning* (PAUL).

The results obtained have been satisfactory because, even though PAUL only tries to approximate ELBG, its performances are very close to those of ELBG and, as we expected, it is more efficient than PARELBG. Besides, PAUL inherits from ELBG its independence of the initial conditions and the high speed of convergence.

The special expedients used for implementing PAUL efficiently on a parallel computing system imply some limitations to its scalability when the number of process it is subdivided into (N) grows. In fact, PAUL has practically the same performances as ELBG when the number of classes (or codewords, N_C) is high with respect to N. While, when it is $\frac{N_C}{N} \leq 2$, PAUL becomes equal to PARLBG with consequent deterioration of the performances.

According to the results obtained and illustrated in Section 9.3, we have the confirmation that PAUL has very complex problems with a lot of patterns and codewords as its ideal sphere of application. However, as will be made clearer in the same Section 9.3, the efficiency of PAUL allows it to outperform other algorithms also for problems falling outside its ideal field of application.

The chapter is organized as follows: in Section 9.2 PAUL is presented and its results and comparisons with other algorithms are reported in Section 9.3; lastly, Section 9.4 reports the author's conclusions.

9.2 The algorithm

9.2.1 General considerations

The fact that the results, in terms of speed-up, were satisfactory for PARLBG and less for PARELBG, shows that the parallel computation of the Voronoi partition, as there implemented, is an effective solution while the serial execution of the ELBG block is a strong limitation for the maximum achievable speed-up. For example, as regards the compression of the image of *Lena* considered in chapter 8, the execution times for the serial version of the algorithm on the MULTISOFT Machine indicate that, on such system, the ELBG-block introduces an overhead of about $10-20\%$ per iteration. Consequently, according to Amdhal's law [98], the maximum achievable speed-up with an infinite number of processors is $5 \div 10$. This theoretical value does not include the time spent in inter-process communications. So, considering the experimental results obtained, we realized that a new implementation of the ELBG block, or an approximated one was necessary. Such an implementation had to allow the parallel execution of the ELBG block without excessively increasing the inter-process communication.

Another problem to be solved regards broadcasting. In PARLBG and PARELBG, the master, at the beginning of each iteration, broadcasts the codebook to all of the slaves. Although we use a network with broadcast functionalities, PVM implements such a function as a sequence of point-to-point transmissions to the slaves. From the point of view of the user interface, the implementation is totally transparent; but it is not the same as regards the performance because the broadcast potentiality of the Fast Ethernet are not used. On the other hand, the management of a reliable broadcast communication is a rather complex problem [99].

Besides, in order to allow PAUL to deal with very complex problems (i.e., with high values for N_P, N_C and k), a policy for the efficient management of the memory has been developed *ad hoc*.

For the problems just reported, and for other considerations that will be presented in the following, several important modifications have been made in PAUL with respect to PARELBG. They can be summarized as follows:

- each task locally performs the ELBG block on a portion of the codewords and without any communication with the others;

- the broadcasting of the codebook has been eliminated. With the new algorithm, each process directly transmits data to each of the other processes. As we will see later, such transmissions do not concern all of the codebook, but parts of it;

- if the physical memory of a host is not enough to store all of the data, the memory necessary to execute the several pieces constituting an iteration is allocated when it is needed and released when it is no longer necessary. Such a solution allows us to work with very complex problems.

9.2.2 The essential points of the new algorithm

In this sub-section we will outline how PAUL works, while, in the following sub-sections, we will progressively explain the details related to the implementation. Like ELBG and PARELBG, PAUL is an iterative algorithm. It is executed in parallel by 1 master and N slaves.

The master

The master deals with the operations related to the initialization and the termination of the algorithm. Besides, at each iteration, it collects from the slaves the data that are necessary to check the termination condition (that is the same as the ELBG). If this is verified, it ends all processes correctly.

The slaves

The slaves all work in the same way and, in the following, we will indicate the generic slave also as *generic task* or, more briefly, as *task*.

Fig. 9.1 outlines how the generic task works. It is possible to distinguish a part related to the initialization (phase F_0) and an iterative part, subdivided again into two phases (F_I and F_{II}). During F_I, the calculation of the Voronoi Partition is executed; during F_{II} the ELBG block is executed and the new centroids are calculated. The last part of both of these phases provides for the sharing of the results with all of the other tasks. The task leaves the iterative phase (and finishes) when the master, once the termination condition is verified, sends the *kill* signal to all of the tasks. More in detail, the operations executed during the different phases, are the following:

- **F_0.** This is the initialization phase. In F_0, the master assigns to ith process a subset (X_i) of the input patterns. The set formed by all of the X_i is a partition of the input dataset. As this operation occurs only at the beginning, we call it *static partitioning.* Moreover, during this phase, the master transmits all of the input patterns to the tasks.

  Besides, each slave computes, autonomously and randomly, the initial codebook. The codebook autonomously calculated by each slave without any interaction is the same for all of them. This is achieved by employing the same technique that will be used for the autonomous selection of the cells (see Section 9.2.3 for details).

- **F_I.** At the beginning of this phase, the ith task, autonomously, selects a subset of the codebook (Y_i) and the related cells. In the following, we will call them local codewords and local cells, respectively. All of

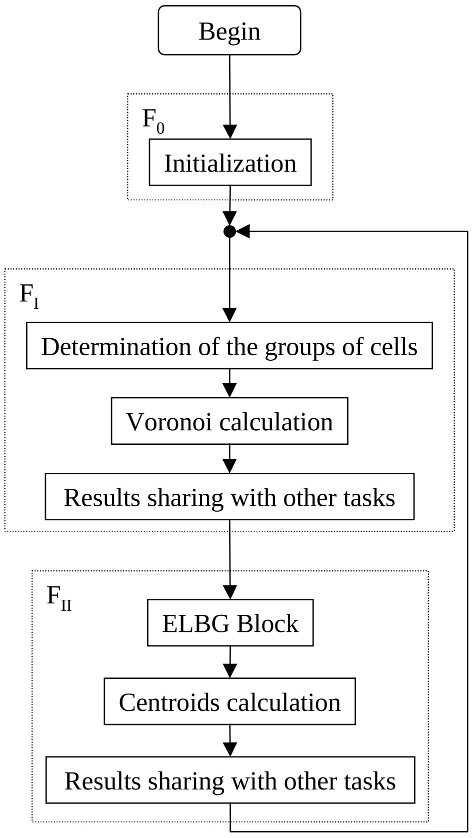

Begin
F0
Initialization
FI
Determination of the groups of cells
Voronoi calculation
Results sharing with other tasks
FII
ELBG Block
Centroids calculation
Results sharing with other tasks

Figure 9.1: Operations executed by the generic task

the Y_i constitute a partition of the codebook. As such a partitioning is different at each iteration, we associate the adjective *dynamic* to it.

Afterwards, for each pattern belonging to X_i, it finds the nearest codeword of the whole codebook (Y). This is the calculation of its portion of the Voronoi partition.

Lastly, it shares the elaborated information with all of the other tasks.

- F_{II}. Beginning from the information just received, the task executes the ELBG block on the local cells . Let us remember that the aim of the ELBG block is the testing (SoCAs) and the eventual execution (SoC) of several shifting of codewords in order to obtain a better distribution for them.

  When the execution of the ELBG block has ended, the ith task calculates the new centroids for the local cells.

Afterwards, it shares the information that it has elaborated with all of the other tasks and goes back to phase F_I.

The execution of phase F_I (calculation of a portion of the Voronoi partition) is exactly the same as it was in PARELBG, while the realization of F_{II} is different. In fact, while in PARELBG, it was entirely (serially) executed by the master, now each process manages a portion of the cells. It executes the SoCAs and calculates the centroids only for its portion. The different execution of F_{II} is the main difference between PAUL and PARELBG.

In PAUL, like in PARELBG, the ELBG-block consists of the realization of several SoCAs and their eventual transformation into SoCs. But, in PAUL, SoCAs occur exclusively between local cells, i.e. cells assigned to the task in question, allowing each task to work in total independence of other tasks. However, such a solution is a limitation of the possibilities of shifting of the codewords. This limitation was absent in the original ELBG and in PARELBG, where SoCAs could occur between all the codewords of the whole codebook. For this reason, in PAUL, a new technique (described later) has been developed for making groups dynamic. So, a cell that at iteration n is assigned to a group (task) , is, generally, assigned to another group (task) at iteration $n + 1$. With this trick, codewords regain a greater freedom of shifting in SoCAs.

The modifications made to the ELBG-block allow it to be efficiently executed in parallel and, thanks to the creation of dynamic groups of codewords, the algorithm obtained is a very good approximation of the original ELBG. As we will see in Section 9.3, the results obtained by PAUL in terms of MQE are very close to the ones obtained by ELBG, where cells are considered all together rather than subdivided into groups. Of course, for $N = 1$, PAUL is equivalent to ELBG.

9.2.3 Determining the portions

In the previous sub-section we said that each iteration of PAUL can be subdivided into two phases. We saw that, during phase F_I, the process considers only a portion of the input patterns while, during phase F_{II}, it considers only a part of the cells. Now, we analyze in detail both of these modes and report some considerations about the problem of the load balancing deriving from the partitioning methods adopted.

- *Determining the input patterns to assign to each process.* Before starting the iterative phase, the master divides the input patterns into N fixed portions and assigns a portion to each of the N processes. Such an assignment is static because once the portion is assigned to a process, it does not change any more. Portions are "fairly" determined by imposing that each subset has the same number of elements. At most, a difference of one element can occur if N_P is not divisible by N.

- *Determining the cells to assign to each process.* In this case we deal with a dynamic assignment because the portions of cells pertaining to each process change with iterations, the number of elements inside each of them being constant. During the initialization, the master assigns, statically and "fairly" (with the same meaning that the term "fairly" has at the previous point), a *number* of cells to each process. This information is transmitted to all of the other processes before the iterative phase starts. So, each process knows how many cells constitute the portions (its own and the others). The fixed assignment concerns only the *number* of the cells constituting each group, while the composition of such groups changes iteration by iteration. For this reason we use the word "dynamic". The generic task chooses the cells constituting its group in an *autonomous*, *random* and *correct* way at each iteration. Now, let us explain the meaning of such properties and see how they have been satisfied.

  - *Autonomous.* Each task determines autonomously its portion of cells and the portions of the other tasks.
  - *Random.* The random choice of the cells allows us to create, in a stochastic way, dynamic partitions constituted by different elements at each iteration. This is useful for realizing the SoCAs in a more efficient way, as explained before.
  - *Correct.* The N portions always constitute a partition of the cells.

  The three properties just mentioned are satisfied by means of a function generating pseudo-random numbers that each task initializes with the same seed as the others. So, all of the tasks generate the same sequence of numbers allowing each of them to determine the portions in an autonomous, random and correct way.

- *Load balancing.* By assigning the same number of input patterns (during phase F_I) and the same number of cells (during phase F_{II}) to each process, we try to effect the load balancing. As regards the (static) partitioning of the input patterns, we, practically, reach a perfect balancing. In the other case (dynamic partitioning of the cells), generally, this is not true because the cardinality of the cells is not constant, but, on the contrary, it can be very different. Nevertheless, as we will see in Section 9.3, the experimental results obtained are interesting.

9.2.4 Full procedure and inter-process communication

In this sub-section, we will detail with some points that we outlined in the previous sub-sections. In particular, we will deal with the inter-process communication both between the master and the slaves and between the slaves.

The master and its interaction with the slaves

During the initialization, the master spawns the tasks on the hosts, reads the configuration and data files, (randomly) initializes the codebook and determines the dimensions of the portions of the learning patterns and of the cells. Afterwards, it transmits these data to the slaves and begins its iterative phase, practically, consisting of the checking of the termination condition. In this phase, the master waits, at each iteration, for the results related to the distorsion from the slaves. Each slave, at the end of the calculation of the Voronoi partition, transmits to the master the total distortion related to its portion of input patterns. The master gets the total distorsion of the global quantizer at mth iteration (D_m) by simply adding the values received from the N slaves. If $\mid (D_{m-1} - D_m) \mid / D_{m-1} \leq \epsilon$, the termination condition is reached and the master terminates the algorithm by killing all of the slaves and saving the results obtained. Otherwise, it waits for the results related to a new iteration.

Communication between the slaves

Now, we will consider the data that are necessary for the tasks to be able to correctly execute the operations constituting phases F_I and F_{II} of an iteration and how these data have to be exchanged if we want to minimize the inter-process communication.

- For the execution of phase F_I, each task needs to access its portion of input patterns and all of the codebook.

  The input patterns are read by the master from a file and are transmitted, once and for all, to the tasks at the end of the initialization operations.

  The initial codebook is randomly and autonomously calculated by all of the slaves. Afterwards, during phase F_{II} of each iteration, they update their portion of the codebook. So, it is necessary that the updatings related to all of the portions are available to all of the tasks. Only in this way, they can correctly execute phase F_I of the following iteration. For this reason, at the end of phase F_{II}, every process communicates to all of the others its updated portion of the codebook. Thus, each process, by putting together all the of the pieces it receives, can obtain the whole updated codebook.

- In order to execute phase F_{II}, each process needs to know:

  1. which cells constitute its portion;
  2. the codewords representing such cells;
  3. which learning patterns constitute each of such cells.

  Regarding point 1, we have already seen previously that each task can autonomously and correctly determine the desired information.

  Data related to point 2 can be accessed by the task without further communications because it keeps in its memory *all* of the codebook, that it used for executing phase F_I.

  On the contrary, the task cannot access all of the data indicated at point 3. Actually, it has in its memory all of the input patterns because they were transmitted by the master before the beginning of the iterative phase. Besides, it knows the portion of the Voronoi partition that it calculated during F_I. Nevertheless, this is not enough for determining which patterns constitute each of the cells to be considered during this phase. This information is "distributed" among all of the tasks given that, during the execution of F_I, each of them calculated a piece of the Voronoi partition.

  A natural way of sharing such information could be that, at the end of F_I, each task broadcasts the results related to the calculation of its piece of the Voronoi partition to all of the other tasks. Nevertheless, we said, previously, that the PVM does not have a real broadcasting function and it effects only a series of identical one-to-one transmissions to the other tasks. As a similar solution would introduce a considerable overhead to the communication process, we adopted a different strategy by exploiting the fact that each task knows its portion of cells and the portions of all the other tasks. So, at the end of F_I, a task does not transmit all of the information related to its portion of the Voronoi partition to all the other tasks. Instead, it transmits a personalized message to each of them containing only that piece of information interests it.

Observation about the inter-process communication

In chapter 5 we saw that, in the serial version of ELBG, during the calculation of the Voronoi partition, several auxiliary arrays and matrices are filled with values that are used subsequently for executing the ELBG block and calculating the new centroids.

How do these data structures have to be filled in PAUL? A possible solution could be to let each task fill a portion of the arrays and matrices and to transmit such data to the other tasks. However, we, experimentally, saw that it is better to transmit only the information related to the partitioning of the input patterns (i.e. the pairs pattern-codeword) and to charge each task with the calculation, from this information, of all the remaining data.

In this way, some more operations are required with respect to the serial version for calculating the auxiliary information. However, considering the bandwidth of our network, the time required for their execution is less than the time required for transmitting such data from one task to another through the network.

9.2.5 Memory management

The management of the physical memory of the single PEs is a very important topic as regards the scalability of the algorithm.

As we saw previously, a task can correctly execute the ELBG block only if it can access all of the patterns constituting the portion of cells assigned to it. Such patterns change iteration by iteration and, so, in order to avoid their retransmission every time, each process receives all of them before the iterative phase starts and stores them either on a file or in its memory.

The choice of storing the patterns either on a file or in the physical memory depends on the related dimensions. In case the latter option is chosen, each PE has to be able to store, besides the above-mentioned patterns, all of the codebook (usually, of substantially smaller size than the input patterns), all of the auxiliary arrays and matrices necessary to store information to correctly execute the ELBG block (see [4]) and all of the data structures necessary for the inter-process communication. Otherwise, the operating system would need to manage the swapping of the memory on the disk with the subsequent collapse of the performance of the whole algorithm [78]. Such a solution, though very efficient from the computational point of view (when enough memory is available), is a strong limitation to the scalability of the algorithm for problems with high values of N_P and k. It would be more suitable if the maximum size of the problem to deal with is determined by the total amount of memory available on the distributed system and not by that available on the single PE.

For this reason, PAUL can work in two distinct modes according to whether the physical memory of each PE is or is not enough to store all of the data. In the former case there is no problem and all of the information necessary to the execution of the algorithm are stored in the physical memory of each PE. In the latter case, a more complex strategy is adopted that, however, allows the utilization of the aggregated resources also as regards the memory. It is based on the use of a local file for the learning patterns, dynamic management of the memory and dynamic loading of the patterns. Now, let us describe the details related to this mode.

1. *Local file for the learning patterns.* Each slave of PAUL stores the input patterns on a proper local file, i.e. a file stored on a local hard disk, that it manages by itself.

2. *Dynamic management of the memory.* At each iteration, the memory to store the data structures for the correct execution of the algorithm [4] and for the inter-process communication is allocated when it is necessary and deallocated when it is no longer needed.

3. *Dynamic loading of the patterns.* At each iteration, the patterns required are loaded from the file to the memory when it is necessary and they are discarded when they are no longer necessary.

Points 2 and 3, at each iteration, introduce an overhead with respect to the situation where the memory is allocated once and for all and all of the patterns are permanently stored. So, this mode is adopted only when the physical memory of the PEs is not enough to hold all the data. In that case, the use of the file of the patterns directly managed by PAUL brings an improvement with respect to the case when the complete management of the memory is left to the operating system. To evaluate this benefit, we have performed a test where the size of the data to be processed is considerably greater than the available physical memory. In particular, using a data set of about 70 MB, PAUL has been launched on a single PE of the MULTISOFT machine that, we must remember, is endowed with 32 MB of physical memory and a local hard disk. This can be used both by the operating system for the management of the swapping of the memory and by PAUL for storing its local file for the learning patterns. If the file for the learning patterns managed by PAUL is adopted, the mean time required per iteration is about 159 s. While, if we try to store all the data in the memory of the PE (with consequent swapping by the operating system), about 257 s per iteration are required.

9.3 Results and comparisons with previous works

In this section we propose the results obtained by PAUL. All of the tests have been executed by using the RMSE as the distortion measure and $\epsilon = 0.001$ as the threshold for the termination condition. Each of the results in this section related to the performance of PAUL is the mean value of 10 runs.

9.3.1 Compression of large-sized images

The first test consists of the compression of a large-sized image. An in-depth analysis of the results obtained is performed as regards both the final quantization error and the computational efficiency (number of required iterations and speed up).

The image we chose is *giraffe*, a 256 gray-level picture of size 984×1488 pixels that we subdivide into 91512 square blocks, each of 4×4 pixels. If we consider each of these blocks as a 16-dimensional vector, we get 91512 16-dimensional input patterns that we use for testing PAUL.

Table 9.1 reports the results when N_C and N vary. The following points are important for the correct interpretation of the values.

- RMSE: it is the final error obtained by the quantizer and is considered as a function of N_C and N ($\text{RMSE}(N_C, N)$).
- ΔRMSE(%): it is the percentual increment of the RMSE with respect to the value we would obtain when using the serial version of the algorithm with the same number of codewords. It is equal to $100 \times \dfrac{\text{RMSE}(N_C, N) - \text{RMSE}(N_C, 1)}{\text{RMSE}(N_C, 1)}$
- N_{it}: number of iterations.
- T: total execution time.
- S: speed up.

Analysis of the final quantization error

The analysis of Table 9.1 shows that ΔRMSE(%) is always below 4% and, for a number of codewords greater than or equal to 64, it is below 1% while, for $N_C = 32$ and $N = 16, 20$, ΔRMSE(%) is quite high. This is because, for such values of N_C and N, SoCAs cannot be executed. In fact, for effecting

N_C	N	RMSE	ΔRMSE(%)	N_{it}	T	S
	1	18.813	0.00	11.2	23.9	1.00
	2	18.744	-0.37	12.1	16.9	1.41
32	4	18.791	-0.12	13.3	13.2	1.81
	8	18.820	0.04	16.6	21.1	1.13
	16	19.523	3.77	31.0	108.3	0.22
	20	19.329	2.74	31.0	142.1	0.17
	1	15.928	0.00	12.3	41.5	1.00
	2	15.969	0.25	12.4	26.3	1.58
64	4	15.937	0.06	13.5	22.7	1.83
	8	16.027	0.62	14.7	34.9	1.19
	16	16.055	0.80	17.8	60.9	0.68
	20	16.068	0.88	17.1	65.3	0.64
	1	13.724	0.00	12.8	70.4	1.00
	2	13.754	0.22	12.5	39.5	1.78
128	4	13.737	0.10	14.5	29.0	2.43
	8	13.768	0.32	13.7	35.9	1.96
	16	13.803	0.58	14.8	53.0	1.33
	20	13.824	0.73	15.4	59.2	1.19
	1	11.973	0.00	12.3	119.8	1.00
	2	11.983	0.08	12.6	65.7	1.82
256	4	11.988	0.12	12.6	41.7	2.87
	8	11.995	0.18	12.6	37.8	3.17
	16	12.010	0.31	13.1	48.5	2.47
	20	12.019	0.38	13.8	55.1	2.17
	1	10.487	0.00	12.2	223.7	1.00
	2	10.492	0.05	12.0	117.6	1.90
512	4	10.498	0.11	12.3	64.5	3.47
	8	10.503	0.16	12.5	42.7	5.24
	16	10.517	0.29	12.6	51.8	4.32
	20	10.524	0.36	13.1	70.4	3.18
	1	9.228	0.00	11.7	413.3	1.00
	2	9.230	0.02	11.6	230.4	1.79
1024	4	9.244	0.17	11.9	140.6	2.94
	8	9.244	0.17	12.0	78.8	5.24
	16	9.247	0.21	12.2	55.0	7.51
	20	9.253	0.27	12.3	55.4	7.46
	1	8.078	0.00	12.1	1612.6	1.00
	2	8.088	0.12	12.0	783.6	2.06
2048	4	8.096	0.21	12.3	401.6	4.02
	8	8.109	0.38	12.0	207.6	7.77
	16	8.115	0.46	12.3	140.1	11.51
	20	8.120	0.51	12.2	135.5	11.90
	1	6.937	0.00	13.6	4060.7	1.00
	2	6.943	0.09	14.3	2138.2	1.90
4096	4	6.958	0.31	14.2	1089.1	3.73
	8	6.967	0.44	14.6	570.8	7.11
	16	6.990	0.77	13.8	323.6	12.55
	20	6.997	0.87	13.2	268.2	15.14

Table 9.1: Performance of PAUL with *giraffe*

a SoCA, three cells are necessary (see Figs. 4.4-4.7). But, for these values of N_C and N, the "fair" distribution of the cells among the tasks gives at most two cells to each of them and no SoCAs can be executed. In such cases, PAUL works like PARLBG. In general, PAUL turns into PARLBG when $\frac{N_C}{N} \leq 2$. However, we must remember that PAUL was developed for complex problems with a high number of codewords. For such problems, even if PAUL is, only approximately, the distributed realization of ELBG, we reach very good results that negligibly differ from the values of RMSE obtained by the original serial algorithm. Moreover, let us observe that, for $N_C = 32$ and $N = 2, 4$, ΔRMSE(%) is negative. This is because, for such a value of N_C, the problem of local minima is not so considerable as for a higher value of N_C and, practically, for $N = 2$ or $N = 4$, PAUL works as well as ELBG. Any small differences can be, mainly, ascribed to the stochastic processes inside the algorithm (initialization of the codebook and selection of the cells for the SoCAs).

In [3] it was highlighted, through several examples, that ELBG is practically independent of the choice of the initial codewords and that it works well also by adopting a technique of random initialization. The results obtained by PAUL (that uses the same technique of random initialization of ELBG), very close to those of ELBG, show that it inherits from this the same insensitivity to the initial conditions.

Analysis of the speed of convergence

In [3] the high speed of convergence of ELBG was highlighted, i.e. the low number of iterations required for obtaining the final error. Now, we will analyze the speed of convergence of PAUL and we will verify that its trend is similar to that of ELBG.

In [3], many comparisons between ELBG and other techniques existing in literature were presented. From a graph reported there, where LBG and ELBG are compared, the error of the latter algorithm decreases fast and, in a few iterations, reaches a value very close to the final result. Quantifying the example cited, the RMSE obtained by ELBG after 3 and 4 iterations is about 5% and 3% respectively greater than the final RMSE.

The analysis of Table 9.1 shows that the number of iterations required by PAUL is very close to those required by ELBG except for the case with $N_C = 32$ and $N = 16, 20$, where the number of iterations required by PAUL is considerably higher than ELBG (31 vs. 11.2). But, as we said previously, in such cases PAUL degenerates into PARLBG. Otherwise, the values are very close to those of ELBG. Besides, we have also verified that the trend of the curves of the error versus the number of iterations is very similar for PAUL and ELBG. For example, Fig. 9.2 reports the typical trends of such curves for a run with $N_C = 1024$ for ELBG (solid line) and PAUL with $N = 20$ (dotted line). Practically, starting from the fourth iteration, the two curves are indistinguishable.

Analysis of the speed up

The last column of Table 9.1 reports the values of the speed up obtained by PAUL. Now, we will discuss them and, besides, we will propose a model that, N_C being fixed, allows us to calculate S versus N.

As we were expecting, the maximum value of the speed up that can be achieved before saturation occurs increases when the complexity of the problem (i.e. N_C, the data set being fixed) increases too. For the most

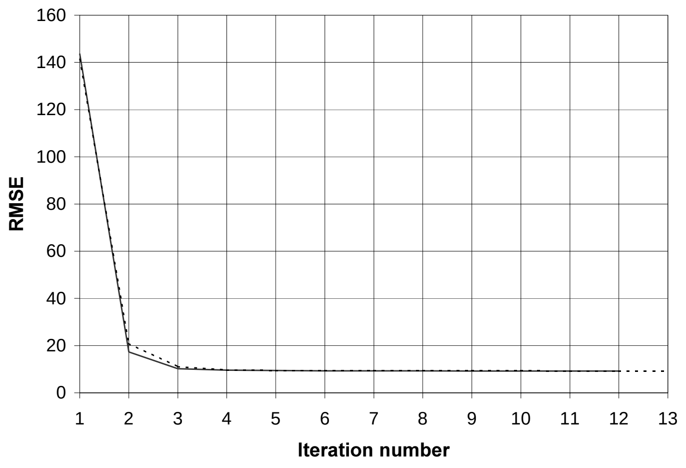


Figure 9.2: RMSE versus iteration number for ELBG (solid line) and PAUL for $N = 20$ (dotted line)

complex problem considered ($N_C = 4096$), for $N = 20$, we obtain a speed up of 15.14 corresponding to a saving in the computation time of about 93% with respect to the serial case.

N_C being fixed, we saw that the trend of S, when N varies, can be well approximated by a law of the kind: $S = aN^m e^{-(bN+cN^2)}$. The values of a, b, c, m can be calculated by considering the logarithm of both of the members of the equation and solving, by a linear least square routine, the system of linear equations obtained substituting the points (i.e. the pairs of values S and N) we wish to approximate.

For example, in Fig. 9.3 we can see the results of the interpolation effected for $N_C = 256$. Because of the good level of approximation we reached in all of the examined cases, we decided to use this method for estimating the maximum value of S achievable also for the cases where the maximum number of available PEs ($N = 20$, in our case) had not produced saturation. In Table 9.2, according to the law just reported, we show, as a function of N_C, the maximum achievable values for S (S_{max}) and the number of processors (N_{max}) for which such a value is obtained.

9.3.2 Comparison with PARLBG and PARELBG

In Table 9.3 we report the comparisons of the results obtained by PAUL with the ones obtained by PARLBG and PARELBG. For the comparison, we used the image of *Lena* [86] of 512×512 pixels. By subdividing it into square blocks of 4×4 pixels, we get 16384 16-dimensional vectors. Because

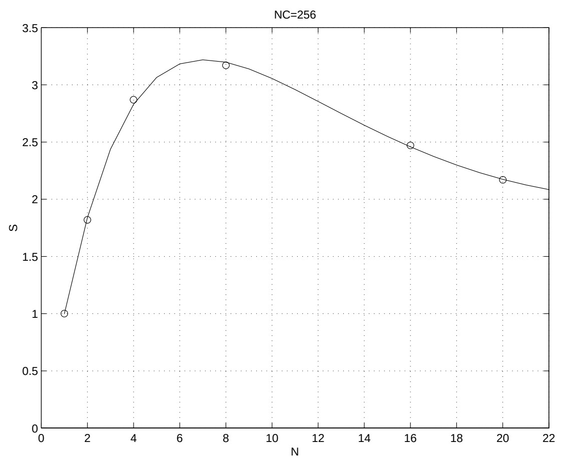


Figure 9.3: Estimate of the speed up for N_C=256

N_C	S_{max}	N_{max}
32	1.84	3
64	1.78	3
128	2.38	5
256	3.21	7
512	5.38	10
1024	7.58	18
2048	11.89	19
4096	21.98	47

Table 9.2: Maximum estimated speed ups vs the number of codewords

the image is not very wide, this is not a very complex task and, so, all of the potential of PAUL cannot be highlighted.

However, we can see that, in terms of speed up, generally, PAUL gets better results than PARELBG and worse than PARLBG.

As regards the quantization error, the performances of PAUL are much better than PARLBG and slightly worse than PARELBG. Again, this is another confirmation that, though PAUL only approximates the operations executed by ELBG, the particular techniques adopted allow it to compete with ELBG.

By analyzing Table 9.3, we can see that there is only a configuration where the speed up of PAUL is worse than PARELBG: $N_C = 1024$ and $N = 2$. First of all, $N_C = 1024$ is the most complex problem we examined

		PARLBG		PARELBG		PAUL	
N_C	N	RMSE	S	RMSE	S	RMSE	S
	1	33.744	1.00	25.815	1.00	25.815	1.00
	2	33.744	1.85	25.815	1.58	25.802	1.81
256	4	33.744	3.44	25.815	2.48	25.815	3.11
	8	33.744	5.83	25.815	3.39	25.861	4.37
	16	33.744	7.51	25.815	4.00	25.908	4.59
	1	31.751	1.00	22.508	1.00	22.508	1.00
	2	31.751	1.95	22.508	1.72	22.557	1.79
512	4	31.751	3.73	22.508	2.83	22.580	3.38
	8	31.751	6.79	22.508	4.18	22.561	5.78
	16	31.751	9.02	22.508	5.32	22.604	6.04
	1	30.517	1.00	19.047	1.00	19.047	1.00
	2	30.517	1.98	19.047	1.84	19.037	1.74
1024	4	30.517	3.89	19.047	2.95	19.047	3.24
	8	30.517	7.17	19.047	4.65	19.101	5.04
	16	30.571	11.24	19.047	6.35	19.101	8.35

Table 9.3: Comparison of the results related to PARLBG, PARELBG and PAUL

as regards the compression of Lena. In that case, also because of the low value of N, the overhead introduced by the serial execution of the ELBG block only slightly affects PARELBG. Such an observation is confirmed by the high value of the speed up obtained by PARELBG ($S = 1.84$), that is very close to the linear speed up ($S = 2$). However, when N increases, the effect of the serial execution of the ELBG block is more and more evident in PARELBG; so, PAUL outperforms PARELBG again as regards the speed up.

9.3.3 Texture segmentation

In [78, 100] an algorithm for large-scale parallel data clustering, named P-CLUSTER, is presented. It is tested for problems of texture segmentation by adopting a technique proposed by Jain and Farrokhnia in [13]. The initial phases of this technique consist of a preliminary filtering of the image to be segmented through a bank of Gabor filters and a subsequent elaboration of the images coming out from the bank. In such a way, a certain number of features are obtained for each pixel. The same authors suggest some criteria for the choice of the number of features to consider. By associating the corresponding vector of the features to each pixel of the image, the input data set used for the final clustering is obtained.

Among the several tests reported in [78, 100], we have considered the ones executed with an image of 512 $\times$ 512 pixels (from the collection of Brodatz [101]) containing 16 distinct textured regions. With the method described above, 20 features per pixel are extracted from it and the patterns so obtained are fed to the clustering algorithm having fixed $N_C = 16$.

As we have seen in Section 9.3.1, the performances of PAUL can be sensibly worse than ELBG when, with such a low number of codewords ($N_C = 16$), a high number of tasks is used. So, we started to compare PAUL and P-CLUSTER when running on a single node of their respective computing system.

The tests in [78, 100] were effected using two different systems:

- a (heterogeneous) collection of Sun Sparc workstations each with 32 MB of memory. With this system, the authors were not able to obtain significant results when using less than 3 hosts. In fact, in such cases, the physical memory available on each of them was not enough to avoid the swapping of the memory on the local hard disk by the operating system with subsequent collapse of the performance of the whole algorithm.
- an IBM SP-2 supercomputer at Argonne National Laboratory. Each node of the SP-2 is equipped with 128 MB of memory. In this case the authors were able to execute the test also on a single machine. For the problem described above, when using a series of techniques for reducing the number of distance calculations, slightly more than 1000 s were necessary.

On the contrary, thanks to the particular expedients for the managing of the memory previously described, the 32 MB equipping a Celeron 366 were enough to make PAUL execute the same test in a total time of 143 s and 7 iterations on a single PE of the MULTISOFT machine. The final result obtained was, practically, identical to the one obtained by P-CLUSTER.

So, in this case, PAUL outperforms P-CLUSTER as regards both the quantity of memory required and the total execution time. We could think that the drop in the execution time is, eventually, due to the increased power of a Celeron 366 with respect to a host of the SP-2 (an IBM RS/6000 model 370 with a clock speed of 62.5 MHz). So, it is better to consider the computational complexity of the two algorithms that, in agreement with the authors of P-CLUSTER, we can identify as the number of distance calculations executed altogether. According to these considerations, we can perform a comparison between PAUL and P-CLUSTER (when running on a single node of the respective computing systems) with the help of the following scheme.

- Number of distance calculations per iteration executed by LBG: $N_P \times N_C = 262,144 \times 16 = 4,194,304$.
- Number of distance calculations executed altogether by P-CLUSTER when none of the techniques for the reduction of their number are used (pre-pruning): 2,926,000 thousand. This value is equivalent to about 698 LBG iterations.
- Number of distance calculations executed altogether by P-CLUSTER when all of the techniques for the reduction of their number are used (post-pruning): 213,400 thousand. This value is equivalent to about 51 LBG iterations.
- As we have previously said, the computational complexity of an ELBG iteration is about $10 \div 20\%$ more than an LBG iteration. So, 7 PAUL iterations (with $N = 1$) are equivalent, by rounding up, to about 9 LBG iterations.

By comparing the 9 LBG-equivalent iterations required by PAUL with the 51 required by P-CLUSTER (post pruning), we can argue that PAUL outperforms P-CLUSTER because of its better efficiency. Besides, thanks to an efficient mechanism for the management of the memory, PAUL overcomes P-CLUSTER also with a quarter of available physical memory.

As regards the speed up achievable for this problem, we performed more tests by increasing the number of processes to a maximum value of $N = 4$. In this case, we got, practically, the same error in 64 s and 8 iterations. The speed up on the total computing time is 2.23. The value of N was not further increased for keeping PAUL far from the threshold $\frac{N_C}{N} = 2$ when it turns into PARLBG.

According to the considerations above, we believe that, also for those problems that are not the ideal field of application of PAUL, i.e. problems with a non-high number of codewords, it constitutes a valid alternative to P-CLUSTER because of its better efficiency. In the case considered, even though PAUL was launched for low values of N (max $N = 4$), it performed better than P-CLUSTER[1] even when this was using 16 processors.

[1]It was rather difficult to quantify better the data related to P-CLUSTER because they have been extrapolated from the graphs of the cited papers.

Besides, by analyzing the tables reported in [78, 100], we can see that, when the input data set is fixed, the number of distance calculations executed by P-CLUSTER does not increase linearly with N_C, but it grows much faster. For this reason, in our opinion, its use is prohibitive for the problems that, on the contrary, constitute the ideal field of application of PAUL.

9.4 Conclusions and future works

In this chapter the authors have described and analyzed PAUL, an algorithm for unsupervised learning that can be efficiently implemented on a parallel computing system. It has been shown that, for complex problems with, at the same time, a high number of patterns and codewords, PAUL obtains good results as regards both the final quantization error and the achievable speed up. Particular optimization techniques allow it to outperform other algorithms for parallel data clustering also when dealing with problems outside its ideal field of application.

Further studies provide for the extension of such a domain to complex problems also with a low number of codewords. Besides, a more advanced strategy of load-balancing will be studied in order to allow the use of heterogeneous computing systems whose hosts have different computing power.

Chapter 10

Conclusions and further studies

In this thesis several new algorithms for Cluster Analysis and Vector Quantization have been presented with the unified denomination of techniques for Unsupervised Learning. The techniques proposed have been classified as belonging to one of the following three cathegories: traditional, incremental and parallel algorithms for CA/VQ. For each of them, besides a detailed description, all of the considerations behind the new ideas have been explained. One of the most important concept developed is the utility of a codeword, that allows the algorithms considered to escape the situations of bad local minima. Particular importance has been given to the the computational efficiency of the algorithms. For this reason, greedy techniques and *ad hoc* data structure have been employed. As regards the results, numerous comparisons with previous existing techniques in literature have been reported and the new algorithms always outperform old ones. Besides, a chapter of the thesis (chapter 7) has been devoted to give a wide description of the MULTISOFT Machine, the computing system we built and used for implementing the parallel algorithms developed. In particular:

- ELBG is a traditional technique for CA/VQ which, being the number of codewords fixed, finds a very good codebook. Its performance are very good as regards both the speed of convergence and the final quantization error;
- FACS is an incremental technique for CA/VQ which, being the target error fixed, finds a codebook that is able to guarantee the desired target error. Its performances as regards the quantization error and the speed of convergence are similar to ELBG;
- PARLBG and PARELBG are a preliminary study about the possibility of implementing parallel algorithms for CA/VQ on the MULTISOFT

Machine. They have given useful indications about which parts of the traditional serial algorithms can be easily implemented on the MULTISOFT Machine and which parts have to be modified. Starting from these indications a new efficient parallel algorithm has been developed: PAUL. It has obtained very good performances as regards both the final quantization error and the speed-up with respect to the serial algorithm of reference, i.e. ELBG.

Further studies will be addressed towards the following directions:

- utilization of different types of distorsions in order to allow a better processing also for linear and elliptic clusters;
- improvement of the proposed parallel techniques, in order to allow the extension of its domain to complex problems also with a low number of codewords;
- development of a more advanced strategy of load-balancing for the proposed parallel techniques in order to allow the use of heterogeneous computing systems whose hosts have different computing power.

Chapter 11

Acknowledgments

I wish to thank Prof. Marco Russo of the University of Messina (Italy) for his collaboration in the activities related to my research and for the valuable suggestions about writing papers that have allowed me to improve considerably the (still bad) quality of my writing. Besides, I wish to thank Andrea Baraldi of the Joint Research Center, Ispra, VA (Italy) for his help in the testing of FACS, in particular for having supplied me with his original code of FOSART. Moreover, thanks to the *Istituto Nazionale di Fisica Nucleare (INFN)* - section of Catania, *Istituto Nazionale di Fisica della Materia (INFM)* - section of Messina and *Centro di Calcolo Elettronico - Università di Messina (CECUM)* for funding the equipments constituting the MULTISOFT Machine. Thanks also to Prof. Salvatore Cavalieri of the University of Catania (Italy) for his cooperation in organizing the activities related to the triennium of my Ph.D. Lastly, thanks to my colleague, Giuseppe Campobello, for sharing with me the numerous extra-research charges of our work and for giving me entertainment with his musical band and thanks to my friend Giuseppe Blancato (called *The Breadkmaker*) for helping me in the selection of the images to elaborate from his huge personal collection.

Bibliography

[1] 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.

[2] G.Patanè and M.Russo, "Comparisons between Fuzzy and Hard Clustering Techniques," in *Proceedings of WILF'99*, June 1999. in press.

[3] M.Russo and G.Patanè, "The Enhanced LBG Algorithm," *Neural Networks*, vol. 14, pp. 1219–1237, Nov. 2001.

[4] G.Patanè and M.Russo, "ELBG implementation," *International Journal of Knowledge based Intelligent Engineering Systems*, vol. 4, pp. 94–109, Apr. 2000.

[5] G. Patanè and M. Russo, "Fully Automatic Clustering System," *IEEE Transactions on Neural Networks*, submitted.

[6] G.Patanè and M.Russo, "Parallel Clustering on A Commodity Supercomputer," in *IJCNN 2000, Proc. of the IEEE-INNS-ENNS Int. Joint Conf. on Neural Networks*, vol. 3, pp. 575–580, 2000.

[7] G.Patanè and M.Russo, "Distributed Unsupervised Learning Using the MULTISOFT Machine," *Information Sciences*, in press.

[8] G.Campobello, G.Patanè, and M.Russo, "PAUL: a Parallel Algorithm for Unsupervised Learning," *Neural Networks*, submitted.

[9] R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis.* New York, NY: John Wiley and Sons, 1973.

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

[11] A. Jain and R. Dubes, *Algorithms for Clustering Data.* Prentice-Hall, 1988.

[12] M. Amadasun and R. King, "Low-level segmentation fo multispectral images via agglomerative clustering of uniform neighbourhoods," *Pattern Recognition*, vol. 21, no. 3, pp. 261–268, 1988.

[13] A. Jain and F. Farrokhnia, "Unsupervised texture segmentation using gabor filters," *Pattern Recognition*, vol. 24, no. 12, pp. 1167–1186, 1991.

[14] H. Frigui and R. Krishnapuram, "A robust competitive clustering algorithm with applications in computer vision," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 21, pp. 450–465, May 1999.

[15] R. Krishnapuram, H. Frigui, and O. Nasraoui, "Fuzzy and probabilistic shell clustering algorithms and their application to boundary detection and surface approximation," *IEEE Transactions on Fuzzy Systems*, vol. 3, pp. 29–60, 1995.

[16] M. C. Clark, L. O. Hall, D. Goldgof, L. P. Clarke, R. Velthuizen, and M. S. Silbiger, "Mri segmentation using fuzzy clustering techniques," *IEEE Engineering in Medicine and Biology*, vol. 13, no. 5, pp. 730–742, 1994.

[17] K.O.Perlmutter, S.M.Perlmutter, R.M.Gray, R.A.Olshen, and K.L.Oehler, "Bayes Risk Weighted Vector Quantization with Posterior Estimation for Image Compression and Classification," *IEEE Transactions on Image Processing*, vol. 5, pp. 347–360, Feb. 1996.

[18] K. Mohiuddin and J. Mao, "A comparative study of different classifiers for handprinted character recognition," in *Pattern Recognition in Practice* (E. Gelsema and L. Kanal, eds.), pp. 437–448, 1994.

[19] J. 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.

[20] R. Baeza-Yates, "Introduction to data structures and algorithms related to information retrieval," in *Information Retrieval: Data Structures and Algorithms* (W. Frakes and R. Baeza-Yates, eds.), pp. 13–27, Upper Saddle River, NJ: Prentice-Hall, Inc., 1992.

[21] S. K. Bhatia and J.S.Deogun, “Conceptual clustering in information retrieval,” *IEEE Transactions on Systems, Man and Cybernetics*, vol. 28, no. 3, pp. 427–436, 1998.

[22] G. Biswas, J. Weinberg, and C. Li, *A Conceptual Clustering Method for Knowledge Discovery in Databases.* Editions Technip, 1995.

[23] C. Carpineto and G. Romano, “A lattice conceptual clustering system and its application to browsing retrieval,” *Machine Learning*, vol. 24, no. 2, pp. 95–122, 1996.

[24] A. Jain, “Data clustering: A review,” *ACM Computing Surveys*, vol. 31, pp. 264–323, Sept. 1999.

[25] T. Bell, J. Cleary, and I. Whitten, *Text Compression.* Englewood Cliffs, NJ: Prentice Hall, 1990.

[26] 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.

[27] T.Lookbaugh, E.A.Riskin, P.A.Chou, and R.M.Gray, “Variable Rate VQ for Speech, Image and Video Compression,” *IEEE Transaction on Communications*, vol. 41, pp. 186–199, 1993.

[28] N. Nasrabadi and R. King, “Image coding using vector quantization: a review,” *IEEE Transaction on Communications*, vol. 36, pp. 957–971, 1988.

[29] E.A.da Silva, D.G.Sampson, and M.Ghanbari, “A Successive Approximation Vector Quantizer for Wavelet Transform Image Coding,” *IEEE Transactions on Image Processing*, vol. 5, no. 2, pp. 299–310, 1996.

[30] 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, pp. 202–225, 1996.

[31] H. Abut, *Vector Quantization.* New York: IEEE Press, 1990.

[32] 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.

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

[34] P. Chou, T. Lookabaugh, and R. Gray, "Entropy-constrained vector quantization," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 37, pp. 31–42, 1989.

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

[36] T.Hofmann and J. Buhmann, "Pairwise Data Clustering by Deterministic Annealing," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 19, pp. 1–14, Jan. 1997.

[37] T. Hofmann and J. M. Buhmann, "Competitive learning algorithms for robust vector quantization," *IEEE Transactions on Signal Processing*, vol. 46, no. 6, pp. 1665–1675, 1998.

[38] 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.

[39] 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.

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

[41] K. S. Al-Sultan, "A tabu search approach to the clustering problem," *Pattern Recognition*, pp. 1443–1451, 1995.

[42] G. C. Osbourn and R. F. Martinez, "Empirically defined regions of influence for clustering analysis," *Pattern Recognition*, vol. 28, no. 11, pp. 1793–1806, 1995.

[43] F. Kowalewski, "A gradient procedure for determining clusters of relatively high point density," *Pattern Recognition*, vol. 28, no. 12, 1995.

[44] A. B. Geva, Y. Steinberg, S. Bruckmair, and G. Nahum, "A comparison of cluster validity criteria for a mixture of normal distributed data," *Pattern Recognition letters*, vol. 21, pp. 511–529, 2000.

[45] H. Frigui and R. Krishnapuram, "Clustering by competitive agglomeration," *Pattern Recognition*, vol. 30, no. 7, pp. 1109–1119, 1997.

[46] A. K. Jain, R. P. W. Duin, and J. Mao, "Statistical pattern recognition: A review," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 1, pp. 4–37, 2000.

[47] V. S. Cherkassky and F. M. Mulier, *Learning from Data: Concepts, Theory and Methods.* John Wiley and Sons, 1998.

[48] A. Baraldi and E. Alpaydin, "Constructive feed-forward ART clustering networks - Part II," *IEEE Transaction on Neural Networks*, in press.

[49] J.C.Bezdek and N.R.Pal, "Two Soft Relatives of Learning Vector Quantization," *Neural Networks*, vol. 8, no. 5, pp. 729–743, 1995.

[50] N.R.Pal, J.C.Bezdek, and R.J.Hathaway, "Sequential Competitive Learning and the Fuzzy c-Means Clustering Algorithms," *Neural Networks*, vol. 9, no. 5, pp. 787–796, 1996.

[51] S. Ahalt, A. Krishnamurty, P. Chen, and D. Melton, "Competitive learning algorithms for vector quantization," *Neural Networks*, vol. 3, pp. 277–290, 1990.

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

[53] 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.

[54] N.B.Karayiannis, "A Methodology for Constructing Fuzzy Algorithms for Learning Vector Quantization," *IEEE Transaction on Neural Networks*, vol. 8, pp. 505–518, May 1997.

[55] 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.

[56] N.B.Karayiannis, J.C.Bezdek, N.R.Pal, R.J.Hathaway, and P.I Pai, "Repairs to GLVQ: A New Family of Competitive Learning Schemes," *IEEE Transaction on Neural Networks*, vol. 7, pp. 1062–1071, Sept. 1996.

[57] A.I.Gonzalez, M.Graña, and A.D'Anjou, "An Analysys of the GLVQ Algorithm," *IEEE Transaction on Neural Networks*, vol. 6, pp. 1012–1016, July 1995.

[58] 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.

[59] 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.

[60] J. McQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, pp. 281–297, 1967.

[61] B. Fritzke, "Vector Quantization with a Growing and Splitting Elastic Net," in *Proceedings of ICANN 93*, 1993.

[62] B. Fritzke, "Fast learning with incremental RBF Networks," *Neural Processing Letters*, vol. 1, no. 1, pp. 2–5, 1994.

[63] B. Fritzke, "Growing Cell Structures - A Self-organizing Network for Unsupervised and Supervised Learning," *Neural Networks*, vol. 7, no. 9, pp. 1441–1460, 1994.

[64] B.Fritzke, "A Growing Neural Gas Network Learns Topologies," in *Advances in Neural Information Processing Systems 7* (G.Tesauro, D.S.Touretzky, and T.K.Leen, eds.), pp. 625–632, MIT Press, Cambridge Ma, 1995.

[65] F. Hamker and D. Heinke, "Implementation and comparison of growing neural gas, growing cell structures and fuzzy artmap," Tech. Rep. 1/97, Schriftenreihe des FG Neuroinformatik der TU Ilmenau, 1997.

[66] T. Martinetz and K. J. Schulten, "A neural-gas network learns topologies," in *Artificial Neural Networks* (T. Kohonen, K. Makisara, and O. Simula, eds.), pp. 397–402, Amsterdam, 1991.

[67] G.A. Carpenter, S. Grossberg, M. Markuzon, J.H. Reynolds, and D.B. Rosen, "Fuzzy ARTMAP: a Neural Network Architecture for Incremental Supervised Learning of Analog Multidimensional Maps," *IEEE Transaction on Neural Networks*, vol. 3, pp. 698–713, 1992.

[68] S. Grossberg, “Adaptive pattern classification and universal recording: I.Parallel development and coding of neural feature detectors,” *Biol. Cybern.*, vol. 23, pp. 121–134, 1976.

[69] A. Baraldi and E. Alpaydin, “Constructive feed-forward ART clustering networks - Part I,” *IEEE Transaction on Neural Networks*, in press.

[70] L. Ni and A. Jain, “A vlsi systolic architecture for pattern clustering,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 7, pp. 80–89, Jan. 1985.

[71] S. Ranka and S. Sahni, “Clustering on a hypercube multicomputer,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 2, pp. 129–137, Apr. 1991.

[72] G. Rudolph, “Parallel clustering on a unidirectional ring,” in *Transputer Applications and Systems* (R. G. et al., ed.), vol. 1, (Amsterdam), IOS Press, 1993.

[73] F. Ancona, S. Rovetta, and R. Zunino, “Parallel architectures for vector quantization,” in *Proc. IEEE Int. Conf. on Neural Networks, ICNN’97*, vol. II, (Houston,TX), pp. 899–903, June 1997.

[74] N.K.Ratha, A.K.Jain, and M.J.Chung, “Clustering using a coarse-grained parallel Genetic Algorithm: A Preliminary Study,” in *IEEE Proc. of Computer Architecturs for Machine Perception*, 1995.

[75] I. Dhillon and D. Modha, “A data clustering algorithm on distributed memory machines,” in *ACM SIGKDD Workshop on Large-Scale Parallel KDD Systems*, Aug. 1999.

[76] B. Zhang, M. Hsu, and G. Forman, “Accurate recasting of parameter estimation algorithms using sufficient statistics for efficient parallel speed-up: Demonstrated for center-based data clustering algorithms,” in *4th European Conference on Principles and Practices of Knowledge Discovery in Databases (PKDD)*, Sept. 2000.

[77] G. Forman and B. Zhang, “Linear speed-up for a parallel non-approximate recasting of center-based clustering algorithms, including k-means, k-harmonic means, and em,” in *ACM SIGKDD Workshop on Distributed and Parallel Knowledge Discovery, KDD-2000*, (Boston,MA), Aug. 2000.

[78] D. Judd, P. McKinley, and A. Jain, "Large-scale parallel data clustering," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, pp. 871–876, Aug. 1998.

[79] "http://www.top-500.org."

[80] "http://www.ieeetfcc.org."

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

[82] I.Katsavounidis, C.-C.J.Kuo, and Z.Zhang, "A New Initialization Tecqnique for Generalized Lloyd iteration," *IEEE Signal Processing Letters*, vol. 1, pp. 144–146, Oct. 1994.

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

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

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

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

[87] A.Teseo and C.S.Regazzoni, "Application to Locally Optimum Detection of a New Noise Model," in *ICASSP'96*, vol. 5, (Atlanta, Georgia), pp. 2467–2470, 1996.

[88] S. Z. Selim and M. A. Ismail, "K-means-type algorithms: A generalized convergence theorem and characterization of local optimality," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 6, no. 1, 1984.

[89] B. Fritzke, "A self-organizing network that can follow non-stationary distributions," in *Proceedings of ICANN 97*, pp. 613–618, Springer, 1997.

[90] B. Fritzke, "Some competitive learning methods," tech. rep., Institute for Neural Computation Ruhr-Universitat Bochum, 1997.

[91] C. M. Bishop, *Neural Networks for Pattern Recognition.* Oxford: Clarendon Press, 1996.

[92] E. B. Baum and K. E. Lang, "Constructing hidden units using examples and queries," in *Advances in Neural Information Processing Systems 3* (R. P. Lippmann, J. E. Moody, and D. S. Touretzky, eds.), pp. 904–910, San Mateo: Morgan Kaufmann, 1991.

[93] K. J. Lang and M. J. Witbrock, "Learning to tell two spirals apart," in *Proceedings of the 1988 Connectionist Models Summer School* (D. Touretzky, ed.), pp. 52–59, San Mateo: Morgan Kaufmann, 1989.

[94] S. E. Fahlman and C. Lebiere, "The cascade-correlation learning architecture," in *Advances in Neural Information Processing Systems 2* (D. S. Touretzky, ed.), pp. 524–532, San Mateo: Morgan Kaufmann, 1990.

[95] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM 3 Users's Guide and Reference Manual.* Engineering Physics and Mathematics Division, Mathematicals Sciences Section, 1994.

[96] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM 3 Users's Guide and Tutorial for Networked Parallel Computing.* Massachussets Institute of Technology, 1994.

[97] F.H.Bennet III, J.R.Koza, J.Shipman, and O.Stiffelman, "Building a Parallel Computer System for $18,000 that performs a Half Peta-Flop per Day," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '99)*, 1999.

[98] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, Jan. 1996.

[99] T. Dunigan and K. Hall, "Pvm and ip multicast," Tech. Rep. ORNL/TM-13030, Computer Science and Mathematics Divsion, Oak Ridge National Laboratory, 1996.

[100] D. Judd, P. McKinley, and A. Jain, "Computational pruning techniques in parallel square-error clustering of large data sets," Tech. Rep. MSU-CPS-96-02, Dept. of Computer Science, Michigan State Univ, East Lansing,Mich., 1996.

[101] P. Brodatz, *A Photographic Album for Artists and Designers.* New York: Dover, 1966.