

# Distributed Evolutionary Algorithm for Vector Quantisation in JAVA<sup>1</sup>

Grzegorz Malewicz

University of Warsaw, Department of Mathematics, Computer Science and Mechanics  
2 Banacha Street  
Warszawa, 02-297  
gm153314@students.mimuw.edu.pl

Władysław Skarbek

Warsaw University of Technology, Institute of Radioelectronics  
15/19 Nowowiejska Street  
Warszawa, 00-665  
skarbek@ire.pw.edu.pl

**Abstract – In this paper we use a modified LBG algorithm for vector quantization. An evolutionary algorithm was developed to find a quantization that has both not too many reference points and small error. Due to the fact that computations of LBG are independent from one another the algorithm could be accelerated by making it parallel and distributed. Experiments show that the algorithm usually finds global minimum.**

## I. INTRODUCTION

This paper deals with the problem of Vector Quantisation (VQ): given a large set  $X=\{x_1, \dots, x_m\}$  of points from  $\mathbb{R}^n$  (a measurement data) we find only a few representatives  $W=\{w_1, \dots, w_k\}$ , called reference points, which form a good approximation of the data set. By "good" we mean that  $X$  is in the "vicinity" of the representatives i.e. the distance from each point from  $X$  to the closest representative is as small as possible. As a matter of fact this problem is known in statistics as cluster analysis and has numerous statistical applications.

Vector quantization is used in many areas of image science [2,3] such as:

- Colour quantisation: we are to reduce the number of colours in an image. To keep image quality close to the original we have to build a colour table which consists of good representatives of the colours from the original colours;
- Image compression: we divide a picture into many uniformly sized adjacent blocks, e.g. 4x4, and treat them as points from  $\mathbb{R}^{16}$ . After quantisation of these points we obtain representatives which will be used in compressed image instead of original squares;
- Image recognition: we are given a video sequence of images e.g. of John Smith. The aim is to find representative images of him. Then we use VQ to obtain them and. Finally and, when a new image is to be recognised, we compare it only with the representatives and not the huge set of images.

## II. LBG ALGORITHM

In this paper we use LBG [4,2] algorithm as a method of vector quantisation. The classic LBG starts with a number of representatives - randomly chosen small fraction of points from the set  $X$ . Then it builds clusters of points from the set around each reference point. A cluster includes points that are close to the representative and distant to other representatives. After the mean  $m_i$  of every cluster has been calculated the means become new representatives  $w_i$ . The process is iterated until the Mean Square Error (MSE) of quantisation becomes stable.

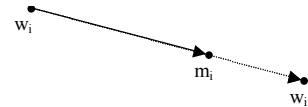


Fig. 1. A reference point  $w_i$  becomes  $w_i'$  instead of  $m_i$ .

Successive Over-Relaxation (SOR) technique was introduced to accelerate classic LBG. It results in an average 20% decrease in the number of iteration LBG has to perform to stabilise MSE. While classic LBG substitutes new means  $m_i$  for representatives, LBG used in this paper overshoots the means:

$$w_i := w_i - sor(w_i - m_i) \quad (1)$$

Experiments show [5] that for uniform measuring sets the parameter  $sor$  should be set to 1.8 for best results. It can be observed though that when measuring set consists of dense "grains" LBG with SOR starts fluctuating. Therefore, it is advisable to set  $sor$  to a smaller value.

A basic drawback of LBG follows from the fact that we have to explicitly set the number of reference points, which the algorithm cannot change. In most cases, though, we do not know what a number of representatives will still make a

<sup>1</sup> This work appeared in the Proceedings of International Conference on Parallel Computing in Electrical Engineering (PARELEC'98), pp. 255-260, Białystok, Poland, September 1998

good quantisation, i.e. a quantisation with small error. Therefore, we seek for an algorithm able to manipulate the size of VQ in search for both small error and small size. One possible solution is to apply an evolutionary algorithm to the problem.

### III. EVOLUTIONARY APPROACH TO VECTOR QUANTISATION

An Evolutionary Algorithm (EA) based on [1] has been developed. The fixed size population consists of chromosomes, each of which represents a set of reference points. The number of reference points may vary from a chromosome to another. For each chromosome a standard quality measure PSNR [2] is calculated to indicate the error of a quantisation.

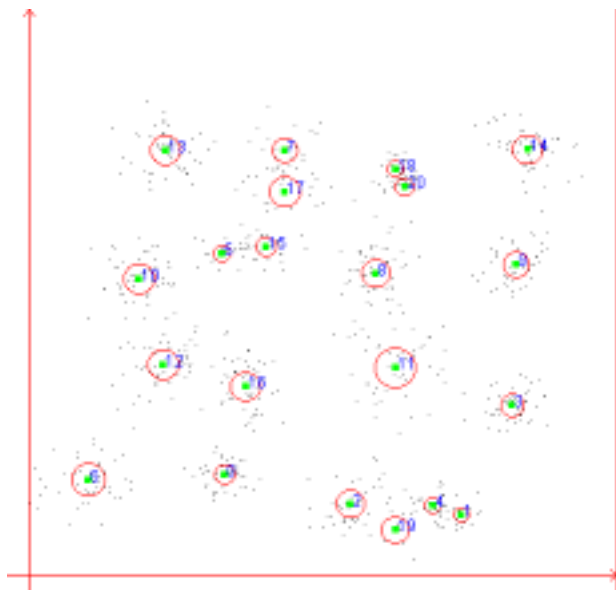


Fig. 2. The population consists of quantizations made of reference points, the number of which can vary from quantization to quantization. Here we can see one such quantization. Circles represent standard deviation of clusters.

The fitness function of a quantization depends on both PSNR and the size of the quantization. The smaller the value of the fitness function the better the quantisation is. The parameter  $\alpha$  indicates how much we want to punish quantisations consisting of many reference points. It is due to us to set it to a desired value.

$$f(W) = \alpha|W| - PSNR(W) \quad , \alpha \geq 0 \quad (2)$$

A special suit of genetic operators has been established. It consists of crossover, mutation, addition and removal. To accelerate searching and to allow for distributing computations classic operators, i.e. crossover and mutation, were redesigned, and the two more were added.

Addition and removal operators were introduced to make EA reach the right size of quantization quicker. As a matter of fact the size of a quantization bears the main role in the

value of its PSNR error. The more reference points a quantization has the lower PSNR is. Though, when the size increases fitness function can also increase, depending on parameter  $\alpha$ . Taking into account these two contrary factors EA should be able to see whether it is worth producing an offspring of bigger or smaller size. It should be able to search for the best size on its own. Therefore, the tendency  $t$  is calculated while EA is running. It tells EA whether to create quantizations of bigger or smaller size (see addition and removal operators).

Crossover differs from the classical crossover operator in a way that it does not produce two offsprings. Instead it creates only one child according to the following rule. A random normalised vector  $v$  is generated, which, attached at the mean EX of the set X, defines hyperplane H orthogonal to  $v$  Fig. 1., which divides the space in two halfspaces. Given two quantisations  $W_1$  and  $W_2$  a sole offspring is made. Reference points of the first parent  $W_1$  lying in the first halfspace combined with points of the second parent  $W_2$  from the second halfspace make reference points of the offspring:

$$\{w \in W_1 : (EX - w) \circ v \geq 0\} \cup \{w \in W_2 : (EX - w) \circ v < 0\} \quad (3)$$

Mutation changes coordinates of reference points of the offspring. A real number can be added with probability  $pMut$  to each coordinate. The probability distribution for this number evolves as the EA is running. First we set the distribution to allow fairly big distortion to be produced. Gradually, the distribution is modified so that smaller and smaller distortions are generated. This scenario is called simulated annealing and is believed to help find global minimum. To get the number we generate a real value with Gaussian distribution  $N(0, Var X)$ , where  $Var X$  is a variance of points of X from EX. Then we divide the value by  $1 + \ln(n)$ , where  $n$  is the number of children which have been born of the population.

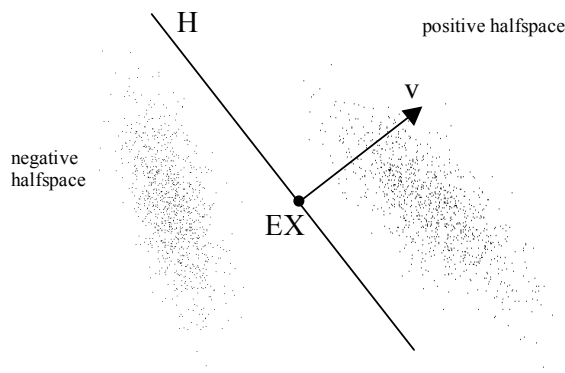


Fig. 3. Hyperplane H divides  $R^n$  into two halfspaces.

The offspring can have some reference points deleted. Although not every child undergoes removal. It happens at random with likelihood  $pRmv-t$ . When it is to take place a real number is randomly chosen from interval  $[0, cRmv-t]$ . It indicates the percentage of reference points to be eliminated. Afterward points are removed in an order. The first one is the

point which has a cluster consisting of the least points from the set X. Then the second smallest is removed, and so on.

At times reference points may be added to the offspring. Similarly to removal, adding happens with probability  $pAdd+t$ . A real number  $a$  is randomly chosen from the interval  $[0, cAdd+t]$ . To evaluate the number of points to be added,  $a$  is multiplied by the number of reference points in the child. New points are randomly selected from the set X.

EA is an iterative algorithm, which seeks the global minimum of the fitness function. The population is initiated with  $popSize$  fixed size quantizations build of points randomly chosen from the set X and tendency  $t$  is set to zero. Then fitness function is calculated for each quantization. Finally evolution begins. It consists of rounds. A round is performed as follows. Two parents are selected according to the roulette rule. They are not extracted from the population, instead clones are created and all the subsequent transformations are exercised with the clones. After crossover operator has been applied, the resulting offspring undergoes addition, removal and mutation in that order. Then LBG is executed on the transformed offspring and fitness function is calculated for the subsequent quantization. Eventually, the quantization is stored in the population only if there is a worse chromosome, which, in such a case, is disposed of. Conversely, when all chromosomes in the population have smaller fitness function than the fitness function of the quantization then the quantization is discarded. In this way the size of the population remains unchanged after a round. Evolution aborts when at least 100 rounds have been carried out and the difference between the fitness function of the best chromosome in the population 40 rounds ago and contemporary the best is below a threshold.

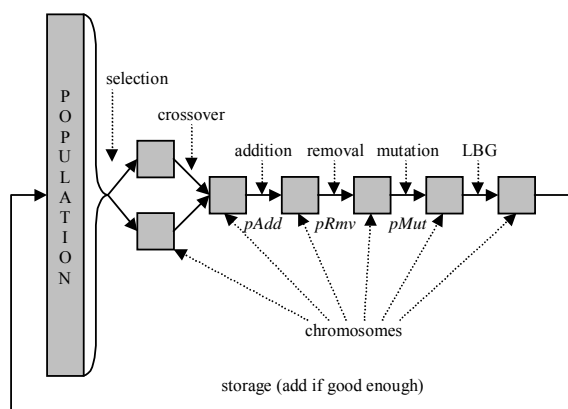


Fig. 4. The sequence of operations in a round.

The tendency  $t$  is refreshed each time a quantization  $W$  is to be stored in the population during evolution stage of EA i.e. at the end of each round. Before the storage the average size  $Avg$  of all quantizations in the population is calculated. If the quantization deserves to be stored then  $t$  changes according to the rule:

$$t := \beta t + \frac{|W| - Avg}{Avg} \quad (4)$$

otherwise it changes as follows:

$$t := \beta t - \frac{1}{2} \frac{|W| - Avg}{Avg} \quad (5)$$

Parameter  $\beta$  is typically set to 0.7. The sign and the value of  $t$  indicate whether it is advisable to produce bigger or smaller offspring. When the population accepts a quantization bigger than the average size  $Avg$  then  $t$  usually increases. On the other hand, if a quantization is smaller then  $t$  decreases.

#### IV. MAKING THE ALGORITHM PARALLEL AND DISTRIBUTED

It is a fact that the iterative process of LBG on an offspring takes some time to compute. For typical data, LBG implemented in JAVA and run on a Pentium 200MHz takes from 10 seconds to a few minutes. The EA has to run LBG about 100 times. Consequently, the whole process can take on one machine from a few minutes to several hours. Yet, the EA allows LBG calculations to be done independently from one another. Therefore, the algorithm can be made parallel and distributed.

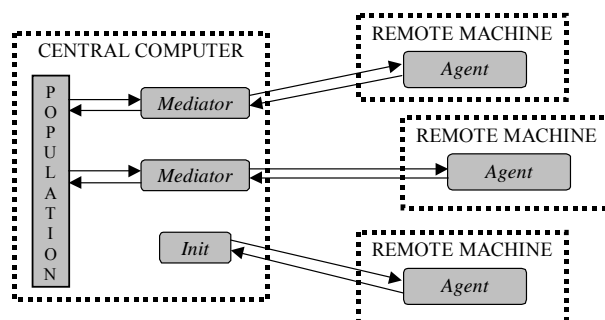


Fig. 5. The structure of communication in distributed EA.

A client-server approach was applied. The population is situated on a central computer. When a remote *Agent* asks for a job, two quantizations are selected from the population and sent away to the machine via the network. After the remote machine has performed crossover, addition, removal and mutation, LBG is carried out and the result is returned to the central computer.

The above scheme was broken into three kinds of processes: *Agent*, *Mediator* and *Init*, see Fig.5. When an *Agent* wants to join computations it contacts *Init* process, which is running on the central computer. *Init* creates a *Mediator* process, which carries out communication between the *Agent* and the population (so each *Agent* on a remote machine has an accompanying *Mediator* running on the central computer). *Mediator* and *Agent* do transferring of data via network. When an *Agent* wants to transmit a quantization to the population it sends it to *Mediator*, which passes it to

the population. Many *Mediators* may want to store quantizations in the population at the same time, hence it is a critical section and monitor can be used to synchronise processes accessing population.

Distributed EA was coded in JAVA language. We will present only the core of the algorithm here, namely class *Population*. The role of an *Agent* and a *Mediator* is clear from Fig. 6. The task of *Init* is to listen on a well-known port for any *Agent* who wants to join computation.

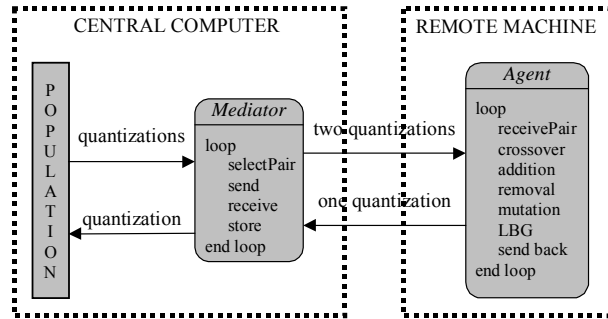


Fig. 6. Data flow diagram and tasks carried out by *Mediator* and *Agent* while performing distributed EA.

Monitor population was implemented as a class *Population* (see Fig. 7.) with *synchronized* methods. JAVA ensures that access of threads to each such method is exclusive, i.e. when each *Mediator* has the same instance of class *Population*, JAVA sees to it that while a *Mediator* runs either *selectPair()* or *store()* methods of the population no other *Mediator* can run them. There are no condition types variables in JAVA. Instead each object has *wait()* and *notifyAll()* methods, first of which suspends a thread whereas the second releases all suspended threads.

The *selectPair()* method, which is called by *Mediators*, returns pairs of quantizations. First *popSize* pairs are created randomly. Consecutive quantizations are acquired by selection, which uses roulette rule. Yet, to ensure enough diversity of genes, it happens only when the population is at least 50% filled. Otherwise a *Mediator* thread is suspended and has to wait until enough quantizations are stored in the population.

When a *Mediator* wants to store a quantization in the population it calls its *store()* method. The population is initially empty and first *popSize* arriving quantizations are stored. When the number of them exceeds *popSize/2* then any *Mediator* thread which has been suspended on *selectPair()* is released. After the number has eventually reached *popSize* (the population is 100% filled with quantizations) the next quantization to be stored is added providing there is a worse quantization in the population (*tryToStoreInPopulation()* method).

There is a slight danger of deadlock in the proposed implementation of distributed EA. Initially, the *Population* creates *popSize* pairs of random quantizations. They are sent to *Agents*, which should return *popSize* quantizations to the *Population*. If half or more quantizations were not stored in the *Population* then no more pairs of quantizations could be

created and *Mediators* would be suspended on *selectPair()* method. Since this scenario is unlikely we do not bother it.

```
class Population
{
    int      popSize;
    Quantization  quant[];
    float    fit[];
    int      created,stored;
    MeasuringData  X;

    synchronized public Quantizations selectPair()
    {
        if( created<2*popSize ) {
            created+=2;
            return( twoRandomQuantizations() );
        }
        else {
            if( stored <= popSize/2 )
                wait();
            created+=2;
            return( selectTwoUsingRoulette() );
        }
    };

    synchronized public void store( Quantization W,
        float fitness )
    {
        stored++;
        if( stored<=popSize ) {
            quant[stored] = W;
            fit[stored] = fitness;
        }
        else
            tryToStoreInPopulation(W,c);
        if( stored>popSize/2 )
            notifyAll();
    }

    public Population()
    {
        stored=0;
        created=0;
        popSize= ... ;
        X= ... ;
    }
}
```

Fig. 7. The core of class *Population*, which synchronises *Mediators'* access to the population.

*Mediator* and *Agent* exchange quantizations through the net. JAVA *Serializable* interface was used to simplify the network protocol, see Fig. 8. A quantization is an instance of *Quantization* class, which inherits from *Serializable* interface. This interface is a standard part of JAVA. Any instance of class that inherits from *Serializable* can be conveniently transferred via network. On the sending side of the connection we create *ObjectOutputStream* of a socket (*getOutputStream()* method) and call its *writeObject()* method. On the receiving side we make *ObjectInputStream* of a socket and call its *readObject()* method. This is basically all we have to do to make an object migrate in the network. It is JAVA job to transmit the object and, recursively, any object that is a part of the object.

```
class Quantization implements Serializable
{ ... }

class Sender
{
```

```

Socket sck;
Quantization quant;
ObjectOutputStream out;

public void send()
    throws IOException
{
    out=new ObjectOutputStream(
        sck.getOutputStream() );
    out.writeObject(quant);
}

class Receiver
{
    Socket sck;
    Quantization quant;
    ObjectInputStream in;

    public void receive()
        throws IOException
    {
        in=new ObjectInputStream(
            sck.getInputStream() );
        quant=(Quantization)in.readObject();
    }
}

```

Fig. 8. An example how to send objects via the network using JAVA Serializable interface, ObjectOutputStream and ObjectInputStream.

## V. EXPERIMENTS WITH MULTIDIMENSIONAL GAUSSIAN DATA

The objective of the test was to establish whether EA was able to find the global minimum of fitness function. To achieve this a relation between the size of the quantization that EA returned and two parameters: the density of grains in the measuring data and the penalty for the size of quantizations was sought.

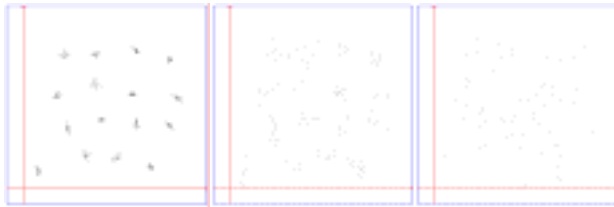


Fig. 9. Measuring data comprised sixteen clouds, which were more and more swollen. Here we see  $X_1$ ,  $X_7$  and  $X_{14}$ .

Fourteen sets  $X_1, \dots, X_{14}$  over  $R^2$  were generated. Each of them comprised Gaussian clouds evenly distributed in the square  $[0,1] \times [0,1]$ , see Fig. 9. (the even distribution was achieved as follows: each cloud was positioned in a  $4 \times 4$  grid, then random distortion was added to each cloud). Any cloud consisted of 50 points and was generated according to  $N(0,1)$  along axis  $x$  and  $y$ . Next it was scaled by  $(\sigma_x, \sigma_y)$  and rotated by a random angle. The standard deviation along each eigenvector of covariance matrix was randomly chosen from  $[0.005, 0.02]$  interval for the set  $X_1$ . It resulted in some clouds being round and some flat. The sets from  $X_2$  to  $X_{14}$  were generated based on  $X_1$ . Each of 16 clouds from  $X_1$  was made more and more swollen. To be precise centroids and eigenvectors of every cloud remained the same, while standard deviation was multiplied by 1.25 (for  $X_2$ ), 1.5 (for

$X_3$ ),  $\dots$ , 4.5 (for  $X_{14}$ ). Consequently, the mean standard deviation of clouds in sets  $X_1, \dots, X_{14}$  equalled 0.0125,  $\dots$ , 0.0531.

The size of population  $popSize$  was set to 8 and initial quantization size was equal to 16. Parameters of genetic operators were set as follows:  $pMut=0.1$ ,  $pAdd=pRmv=0.4$ ,  $cAdd=cRmv=0.3$ . LBG stopped when PSNR started dropping slower than 0.01 per iteration and  $sor$  was set to 1.0. EA was aborted after at least 100 iteration with the threshold set to 0.0p5.

Distributed EA was run in fourteen stages with the measuring data fixed consecutively at  $X_1, \dots, X_{14}$ . At a stage EA was executed 82 times with the parameter  $\alpha$  changing from 0.06 to 2.1. Every time EA looked for a quantization with possibly the smallest value of fitness function. We will discuss the results in two parts. The first includes  $\alpha \in [0.06; 1.067]$ , see Fig. 10., the other  $\alpha \in [1.093; 2.1]$ , see Fig. 11.

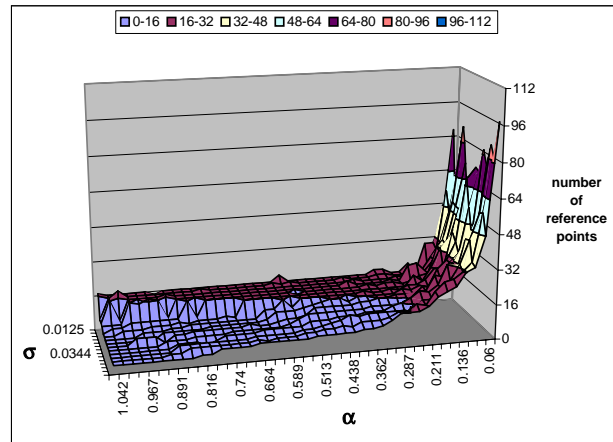


Fig. 10. The results of EA vary according to the value of penalty parameter  $\alpha$  and the degree of concentration of grains in measuring data  $\sigma$ .

We can see in Fig. 10. that when  $\alpha$  increases the number of reference points decreases. It is clear that the bigger the penalty for the size of quantizations the smaller the quantization returned by EA should be.

Let us notice that starting from  $\alpha=0.161$  there is an explosion of number of reference points. The hyperbolic shape of the surface results from the form of formula (2). Namely, after  $\alpha$  has been reduced two times the size  $|W|$  can be increased twice without increasing the value of fitness function.

There are more subtle shapes of the surface. It does not always happen that the outcome of an increase of  $\alpha$  is a decrease of the number of reference points. This is true for swollen clouds (e.g.  $\sigma=0.05$ ) but not for shrunk clouds (e.g.  $\sigma=0.013$ ). Despite the fact that the penalty for the size of quantization increases EA does still return a quantization consisting of 16 reference points (see the triangular region on the surface), which were centroids of clouds. This is not surprising when we recollect the fact that sets  $X_i$ , for small  $i$ , were build of 16 Gaussian clouds distant from each other. In [5] it was proved that for such a measuring data, global

minimum of PSNR in the class of up to 16 reference points quantizations is obtained by 16 points centroidal quantization and any 15, or less, points quantization has sufficiently smaller PSNR. Therefore, even if we increase the penalty, the loss of quality of 15, or less, quantizations is high enough and, consequently, the 16-point centroidal quantization remains the global minimum. We conclude that for big enough  $\alpha$  the decrease of  $\alpha|W|$  when changing from 16 to 15, or less, points quantization will be high enough to outnumber the loss in PSNR. We can see it in the Fig. 10. For given  $\sigma$  when we increase  $\alpha$  (follow to the left in the picture) the number of reference points eventually drops.

Fig. 11. is considerably different from Fig. 10. For  $\alpha$  greater than 1.093 EA does not find 16 points centroidal quantization any longer (we still have some peaks, which result from the fact that EA is a random algorithm). We can see two distinct plateaux. The first refers to four-point quantization. The reason why EA returns such a quantization is similar to the case of 16-point quantization. We only need to treat four adjacent clouds as one, see Fig. 12. The second relate to three-point quantization and can be explained in the same way.

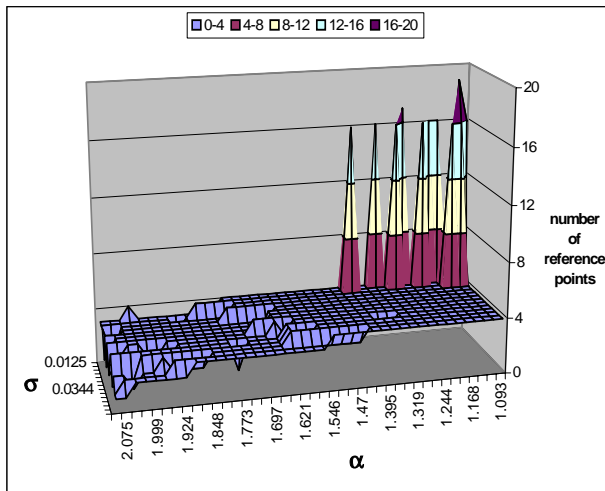


Fig. 11. The results of EA vary according to the value of penalty parameter  $\alpha$  and the degree of concentration of grains in measuring data  $\sigma$ .

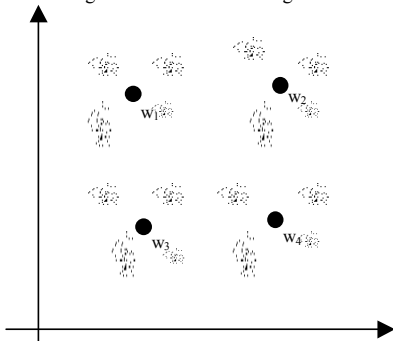


Fig. 12. We can treat each four adjacent clusters as one and put reference points in the middle of them.

## V. CONCLUSIONS

The results of testing distributed evolutionary algorithm for vector quantisation seem very promising. The EA finds the areas of condensation of measuring set. Therefore, it can be used for cluster analysis of subsets of  $R^n$ .

## REFERENCES

- [1] Z. Michalewicz, Genetic algorithm + Data structures = Evolutionary Program, Springer Verlag, 1997.
- [2] W. Skarbek, Methods for representation of digital images, AOW PLJ (in Polish), 1993.
- [3] W. Skarbek (eds.), Multimedia - Algorithms and standards for compression, AOW PLJ (in Polish), 1998.
- [4] Y. Linde, A. Buzo, R. M. Gray, "An Algorithm for Vector Quantizer Design", IEEE Trans. Comm. COM-28, pp. 28-45, 1980.
- [5] G. Malewicz, Distributed Evolutionary Algoriyhm for Vector Quantization, Master Thesis (in Polish) Department of Mathematics, Computer Science and Mechanics, University of Warsaw, 1998.