Technical Report

Yegang Tao

1. Introduction

In rate-distortion theory it has been shown that vector quantization (VQ) outperforms its counterpart, scalar quantization (SQ), consistently [Shannon, 1959], [GRAY, 1998]. However the major problems that may prevent its wide acceptance in image compression applications is the computational complexity. The VQ compression system is highly asymmetry in that significantly more computing time is required for training and encoding than for decoding. This is because the training and encoding process involves many comparisons to look for the best representatives or codewords, whereas the decoder simply hashes the decoded vector according to the index of a codeword specifying its position in the codebook.

Most of the VQ training algorithms, like LBG [Linde, 1980], are based on Generalized Lloyd Algorithm (GLA), a method that iteratively updates the partitions and the corresponding codewords [Lloyd, 1982]. It guarantees to find codewords as the locally optimum solution to the training set. The encoding involves exhaustively search the codebook to find the codeword having minimum distortion to the input vector. So its low distortion is achieved with the expense of very high computation complexities in both training and encoding.

Therefore people were encouraged to investigate on fast design algorithms. They are either using non-iterative clustering techniques [Equitz, 1989]; or by training on a subset to find the codeword rather than the whole set, like tree-structured VQ (TSVQ) [Buzo, 1980]. Although they cannot guarantee to find the optimum solution and may suffer from some degradation of quality, their much reduced complexities make them a good candidate in many real-time image compression applications, for example real-time video coding [Cockshott, 1999]. Moreover in the latter category codewords are usually organized in a well-structured form in contrast with unstructured codebook in GLA. Fast encoding can be benefited from such structures while with some penalties that it cannot guarantee to find the 'closest' codeword to the input vector.

In this report we proposed two new VQ training algorithms. Both generate a codebook by partitioning the hyper-dimensional vector space into sub volumes and choosing the probability weighted centroid of these sub volumes as the code vector (codeword). Both search the codebook using a binary tree. One uses partitioning by hyper-planes orthogonal to the axes of vector space (OAVQ), the other uses

partitioning by hyper-planes orthogonal to the primary principal component (PPC) of the data set (OPVQ).

2. New Fast Vector Quantizer Design Algorithms

2.1 OAVQ

In OAVQ, the partitioning hyper-planes are always passing through the mean of the distribution of training set in current sub volume, and orthogonal to the axes of vector space, along which the distribution of training data in current sub volume has the greatest variance. Every 'splitting' operation will divide current sub volume into two new sub volumes, and we repeat the 'splitting' recursively on each sub volume until we obtain the same number of sub volume as that of codeword we desired, and using the centroid of training data in each sub volume to represent a codeword. Like TSVQ, the encoding process is to determine which sub volume an input vector x should belong to, and use the codeword associated to this sub volume to represent x. A balanced binary tree is used for simplifying the encoding process. The algorithm of OAVQ design and the growth of encoding tree can be described as follows (see Fig. 1).

Algorithm 1: OAVQ

Step 1. Initialization: Set the Tr as the training set and $Tr \in R^K$, a K-dimensional vector space. Set the global variable index of codeword ioc = 0; Set the depth of recursive $dor = \log_2 N$, where N is the number of desired codeword in codebook; set the current tree pointer ctp = *root of encoding tree*.

Step 2. If dor = 0, then compute the centroid of Tr associated with current sub volume and use it to represent a codeword which the index in codebook is ioc. Store ioc in the tree node pointed by ctp. Increase the ioc by 1. We trace back the recursive process to upper level.

Step 3. Determine the means $\{m_i, i = 0,1,\ldots, K-1\}$ of distribution of Tr in current sub volume along each axis.

Step 4. Determine the variances $\{v_i, i = 0,1,\ldots, K-1\}$ of distribution of Tr in current sub volume along each axis.

Step 5. Partition the current sub volume into two (we name these as lower sub volume and upper sub volume for convenience), using a hyper plane normal to the axis of greatest variance passing through the mean along this axis. The hyper-plane can be expressed as: $H(x) = \{x \in R^K \mid (x-a) \cdot b = 0$, where a and b are vectors that a is the mean of distribution and b is the one of the basis of R^K, along which v_i has the greatest value: $b = (b_0, \ldots b_j, \ldots, b_{K-1} \mid \begin{matrix} b_j = 1, \ j = i \\ 0, \ otherwise \end{matrix})$, and a

Store the index of axis $ioa = i$ and the mean of this axis m_i in the tree node pointed by *ctp*.

Step 6. Set $ctp = ctp \rightarrow left$. Decrease *dor* by 1. Perform Steps 2 - 5 recursively on the lower sub volume.

Step 7. Set $ctp = ctp \rightarrow right$. Decrease *dor* by 1. Perform Steps 2 - 5 recursively on the upper sub volume.

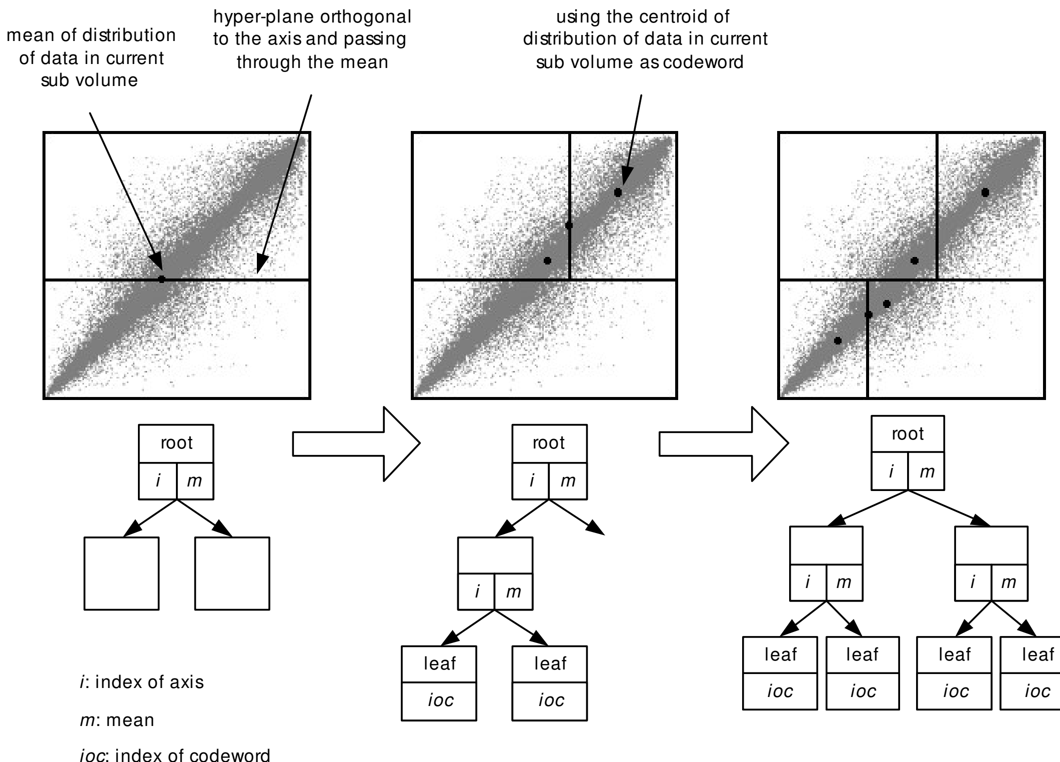

Figure 1: Diagram of OAVQ design on 2-D vector space and growth of its encoding tree.

Fig. 2 gives an example of encoding a 4-dimensional vector x using OAVQ having 8 entries with its encoding tree. Totally, only $\log_2 8 = 3$ times comparisons are performed before we find the best matched codeword. The encoding complexity is linear to the $\log_2 N$ only, no relation to the dimensionality of input vector. This makes encoding performed much faster than the unstructured VQ and even most of the structured VQ, e. g., TSVQ. The sub volumes generated by OAVQ are always hyper-cuboids, such a shape doesn't ensure the reproduction vector is the closest to the input vector, therefore the speed is gained in the expense of degradation of quality.

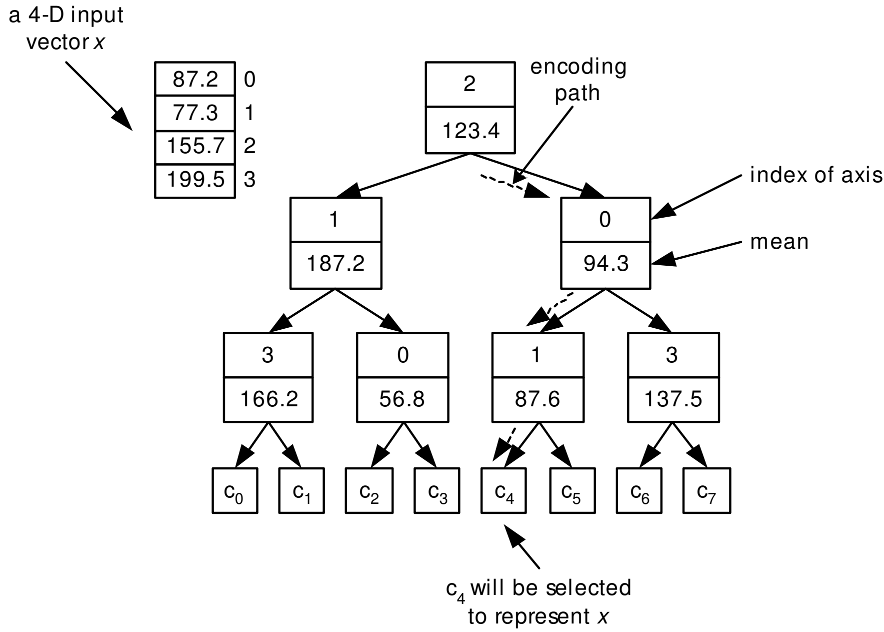

Figure 2: An example of encoding 4-D vector using OAVQ.

2.2 OPVQ

From the previous discussion, we know that vectors sampled from images present a strong tendency for occurrences to be clustered along the diagonal of the vector space. If we partition the vector space using hyper-planes, the hyper-planes normal to the direction of this tendency and passing through the centroid of data set, we ensures that the split contributes the greatest amount to the minimisation of MSE. *Principal component analysis* (PCA) is used for determining the direction of this tendency, and the *first/primary principal component* (PPC) of the data set will be consistent with the direction. The encoding performed similar to OAVQ by determining with sub volume the input vector x should belong to, and using the centroid of training data in this sub volume to represent the x. A balanced binary is designed for encoding. We describe the algorithm of OPVQ as follows (see Fig. 3).

Algorithm 2: OPVQ

Step 1. Initialization: Set the Tr as the training set and $Tr \in R^K$; Set the global variable index of codeword ioc = 0; Set the depth of recursive $dor = \log_2 N$, where N is the number of desired codeword in codebook; set the current tree pointer ctp = *root of encoding tree*.

Step 2. If dor = 0, then compute the centroid of Tr associated with current sub volume and use it to represent a codeword which the index in codebook is ioc. Store ioc in the tree node pointed by ctp. Increase the ioc by 1. We trace back the recursive process to upper level.

Step 3. Compute the centroid and the *weight vector* (PPC) of Tr associated with current sub volume.

Step 4. Partition the current sub volume into two (we name these as lower sub volume and upper sub volume for convenience), using a hyper plane normal to the weight vector *wv* passing through the centroid *cv*. Store the *cv* and the *wv* in the tree node pointed by *ctp*.

$$v \in \begin{cases} Upper\ volume, & f(v) > 0 \\ Lower\ volume, & f(v) \le 0 \end{cases}$$

$$f(v) = wv \cdot (v - cv), \quad \begin{matrix} wv : weight\,vector \\ cv : centroid\,vector \end{matrix} \tag{1}$$

Step 5. Set $ctp = ctp \rightarrow left$. Decrease *dor* by 1. Perform Steps 2 - 4 recursively on the lower sub volume.

Step 6. Set $ctp = ctp \rightarrow right$. Decrease *dor* by 1. Perform Steps 2 - 4 recursively on the upper sub volume.

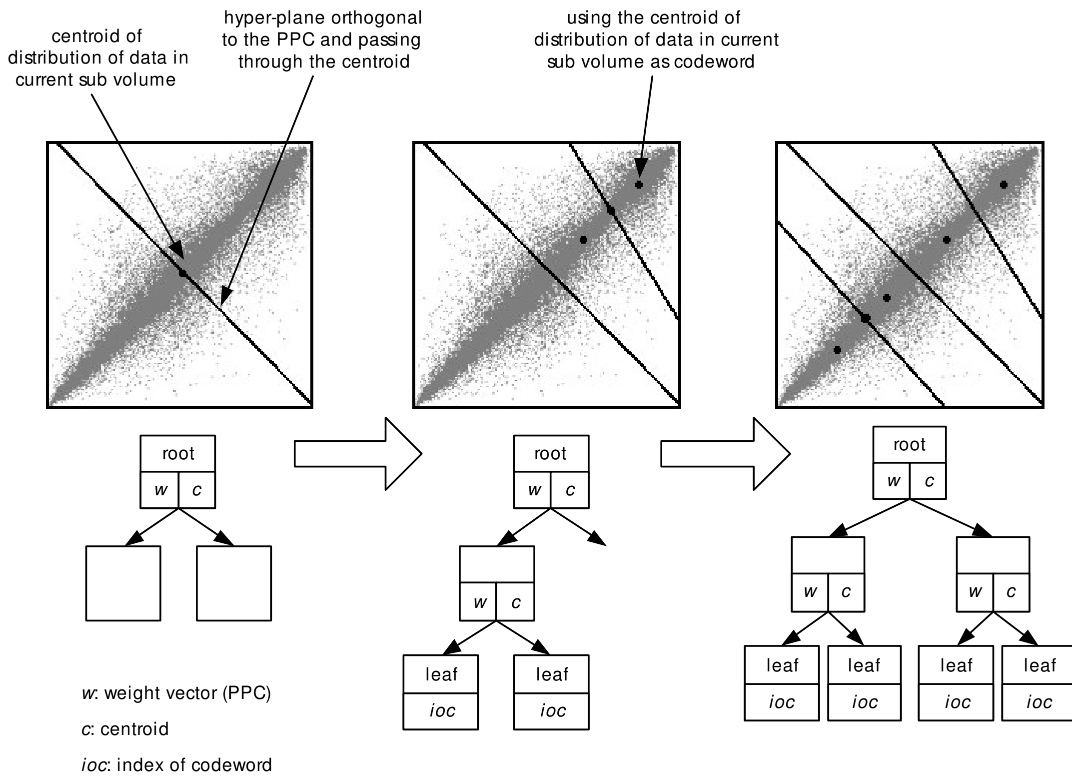

Figure 3: Diagram of OPVQ design on 2-D vector space and growth of its encoding tree.

There are many methods for computing the principal components. Here we use singular value decomposition (SVD) technique [SVQ, Web] to find the largest eigenvalue of the covariance matrix of training set, and use its corresponding eigenvector as the primary principal component. We have noticed that the shapes of

sub volumes generated using OPVQ are very similar to those using TSVQ. Theoretically, in the case of using squared error as distortion measure, the shapes of sub volumes using OPVQ and TSVQ should be same, since both of them use hyper-planes for partition the sub volumes with the purpose of minimizing the average distortion error.

In OPVQ, when encoding, one can using (1) to determine which sub volume an input vector $x \in R^K$ should belong to, and the computational complexity is about $2\times K$. While, in TSVQ, it is determined by comparing the distances of x to the two internal vectors, and the complexity is about $6\times K$.

2.3 Conclusion

We made a comparative study of OAVQ and OPVQ with LBG and PHVQ [Cockshott, 1999] on two aspects: computational complexity of encoding and rate-distortion performance of vector quantizers.

A. Complexity Analysis of Encoding

Assume that we have trained vector quantizers using these four algorithms discussed above. The dimension of the codeword is 2^l and the number of codebook entries is 2^m. We use these four quantizers to compress a picture of dimension x, y which contains $xy/2^l$ vectors. For each vector v, we have

$$v = (v_0, v_1, ..., v_{2^l-1}) .$$

In LBG, we use exhaustive codebook searching method to get the best representative codeword for an input vector. We compute the order of LBG through following steps:

a. The order of computing the square distortion error between two vectors according to Formula 4 is 2^l.
b. The order of finding the best representative codeword is 2^{l+m}.
c. The overall order of compressing a picture using LBG is $\boldsymbol{2^m xy}$.

In PHVQ, we use l lookup tables to map a given vector to the index of representative codeword in codebook. We compute the order of HVQ through following steps:

a. The order of each lookup operation is 1.
b. For each vector of dimension 2^l, the lookup operation will repeat 2^l-1 times.
c. The overall order of PHVQ is $\boldsymbol{(2^l-1)xy/2^l}$.

In OAVQ, we use a balanced binary tree to get the index of representative codeword for a given vector. We compute the order of OAVQ through following steps:

a. For each node in searching tree, we compare the v_{ioa} with *mean* (refer to Fig. 2) and determine which sub-tree we should traverse next. This step is of order 1.

b. The searching tree has the depth of *m*, and the Step a will repeat *m* times before we get the index of codeword to the given vector. This step is of oreder *m*.
c. The overall order of the OAVQ is thus $mxy/2^l$.

In OPVQ, the traversing step of encoding tree is same as OAVQ, but there is more computation for each determination:

a. For each node in searching tree, we compute the $f(v)$ using Formula 17 for a given vector v, and compare $f(v)$ with 0 to determine which sub-tree we should traverse next. This step is of order 2^l.
b. The searching tree has the depth of *m*, and the Step a will repeat *m* times before we get the index of codeword to the given vector. This step is of order $2^l m$.
c. The overall order of the OPVQ is thus mxy.

Table I lists the experimental results of the CPU time for each algorithm on a 1 GHz Pentium III. The source images are 512 by 512 pixels. We use the codebook of 256 entries and 4 by 4 – 16 dimensions of each codeword.

Table I: Average CPU time taken on 30 sample images.

Algorithm	CPU time (μs/vector)
OAVQ	0.76
PHVQ	1.62
OPVQ	34.62
LBG	175.57

B. Rate-distortion Performance Comparison

We examine the performance of these VQs by generating the codebook from a large set natural images and using 'Lena' as the test image ('Lena' is not in the train set). In this experiment VQs were designed as fixed-rate quantizers without entropy coding followed. Table II lists the compression of 'Lena' using these four VQs. We get the compression ratio of 1 bbp (bit per pixel) using codebooks of 256 entries and 8 dimensions, and 0.5 bbp using codebooks of 256 entries and 16 dimensions.

Table II: PSNR results for 'Lena' using four kinds of VQs.

Ratio (bpp)	LBG	OPVQ	OAVQ	PHVQ
1	32.29	31.03	30.52	30.66
0.5	30.35	29.76	28.52	27.77

To obtain detailed results of comparison of LBG, PHVQ, OAVQ and OPVQ, we get results by designing VQs of different codebook entries and dimensions. The codebook entries are ranged from 2 to 256, it can be described as 2^m, where $m = 1, 2, \ldots, 8$. The codebook dimensions are ranged from 4 to 64, it can be described as 2^l, where $l = 2, 3, \ldots, 6$. Fig. 4 demonstrates the surface of PSNR to m and l.

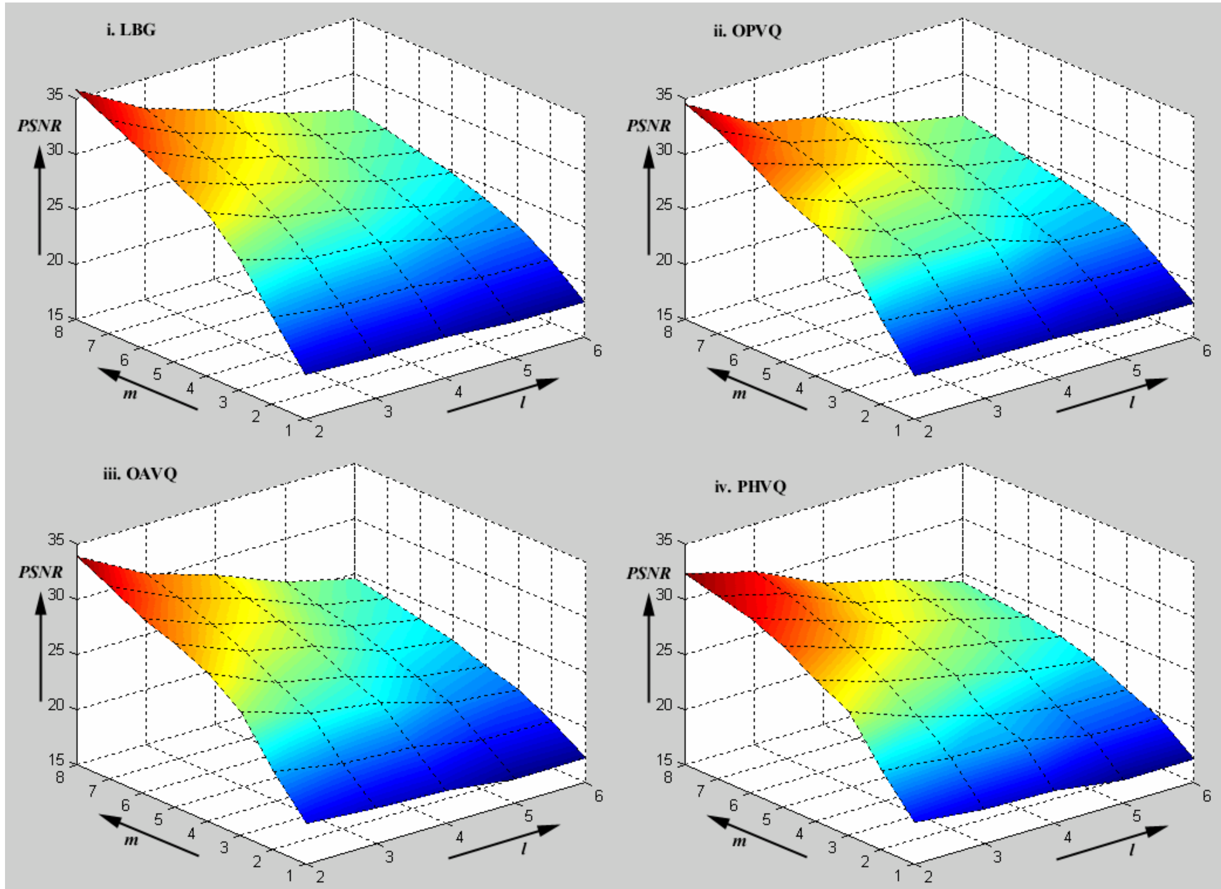

Figure 4: Surfaces of PSNR to m and l of each VQ.

We drew the conclusion from the experimental results. OAVQ has the advantage of low computational complexity in quantizer design and encoding. It can be used as a front end of other VQs, which can simplify their codebook design or encoding procedure. For example, we use OAVQ to produce an initial codebook for LBG, and it makes LBG converged in fewer iterations. In Fig. 5 we can see that codebooks generated by OPVQ are same as those by TSVQ, which are very close to those of unstructured VQ, e.g., LBG. The use of binary tree for encoding makes OPVQ much faster than unstructured VQ, and computation using (1) has lower complexity than computing the distortion measure between input vector and internal vector in TSVQ and therefore makes OPVQ more efficient. It worth emphasizing that both OAVQ and OPVQ can be designed for progressive transmission, like TSVQ. For example, in OAVQ, in addition to storing *ioc* and *mean* in the internal nodes, we store the

centroids of the distribution of train data in the sub volumes which are associated with each internal node (refer to TSVQ).

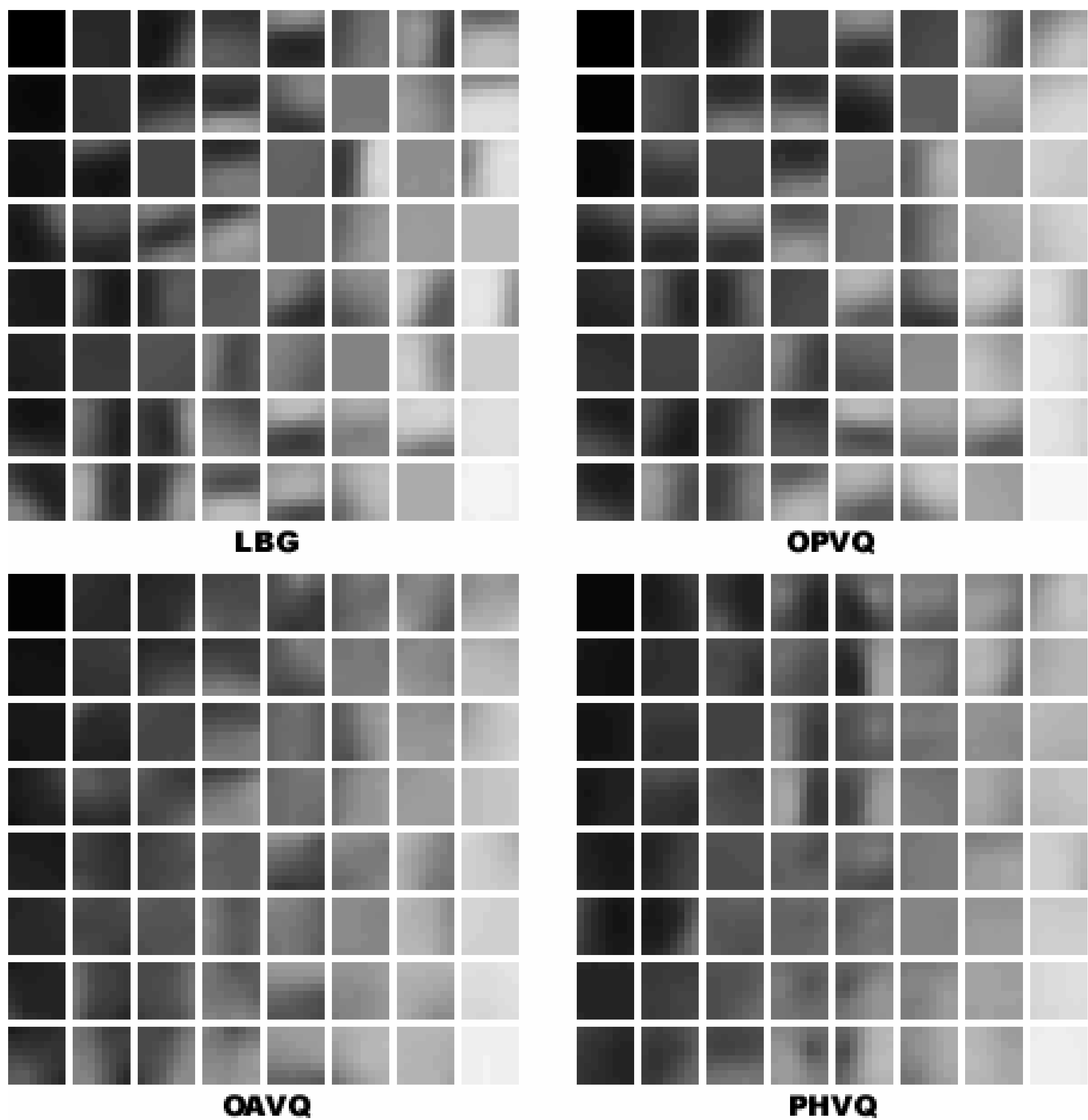

Figure 5: The comparison of codebook patterns with 8x8 block and 64 entries generated by the four VQs.

3. Fast Encoding Algorithm of Unstructured Codebook

We also proposed an algorithm for speeding up the encoding process of unstructured codebook. This method uses effective 'kick out' operations to reduce the number of codeword should be compared before we obtain the best codeword with the minimum distortion error to the input vector.

3.1 Implementations

Algorithm 3: Fast Encoding of Unstructured Codebook (FE-UC)

A. Pre-processing

The Euclidean distances between all paires of codevectors are precomputed and stored in a table. In practice, for each codeword *c* in the codebook, we create a sorted array. We store the index (*idx*) and the distance (*dis*) of other codewords to *c*, from near to far. The 'distance' we mentioned about is defined as squared root distance. Suppose we have a codebook of 256 entries, we'll get a 2D array of 256 rows and 255 columns after pre-processing (See Fig. 6).

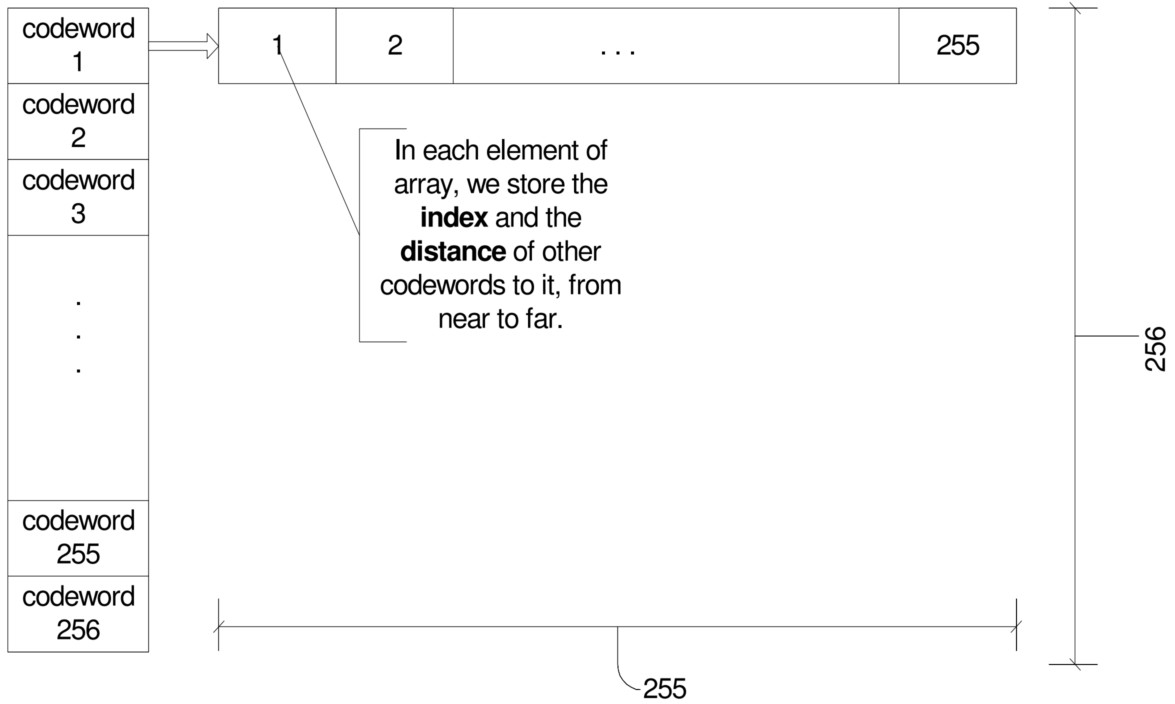

Figure 6 FE-UC: pre-processing of unstructured codebook

B. Fast Searching of the Codebook

Step 1. Given an input vector *v*, and an initial codeword *s* assumed to be the closest codeword to *v*. If $d(v,s) \le \frac{s[1].dis}{2}$, where $s[1].dis$ is the distance of closest codeword to s computed in the pre-processing (refer to Fig. 7 'Condition 1'), s is then the closest codeword we found, and we terminate the searching process; otherwise, we initialise the searching procedure by setting:

i. $c_{closest} = s$: the closest codeword to *v* we found so far.

ii. $c_{ready} = s[1]$: the ready to compared codeword, $s[1]$ is the first element of the array associated with *s* (refer to 'Pre-processing').

iii. $i = 1$: index.

Step 2. Compute the square root distance of c_{ready} to v.

Step 3. If any of the below 3 conditions is satisfied, we terminate our searching, we eventually found the $c_{closest}$ as the closest codeword to v; otherwise, continue to Step 4.

Termination conditions (refer to Fig. 7):

i. The distance of c_{ready} to v is not more than the half of distance of c_{ready} to c_{ready}'s closest codeword:

$$d(v, c_{ready}) \le \frac{c_{ready}[1].dis}{2} \quad (2)$$

ii. The distance of c_{ready} to $c_{closest}$ is greater than the 2 times of distance v to $c_{closest}$:

$$d(v, c_{ready}) > 2 \times d(v, c_{closest}) \quad (3)$$

iii. c_{ready} is the last element of the array associated with $c_{closest}$.

Step 4. If $d(v, c_{ready}) < d(v, c_{closest})$, we update:

i. $c_{closest} = c_{ready}$

ii. $c_{ready} = c_{closest}[1]$

iii. $i = 1$;

otherwise, continue to Step 5.

Step 5. Set c_{ready} is the next codeword of the sorted array associated with $c_{closest}$:

i. $i = i+1$

ii. $c_{ready} = c_{closest}[i]$,

back to Step 2.

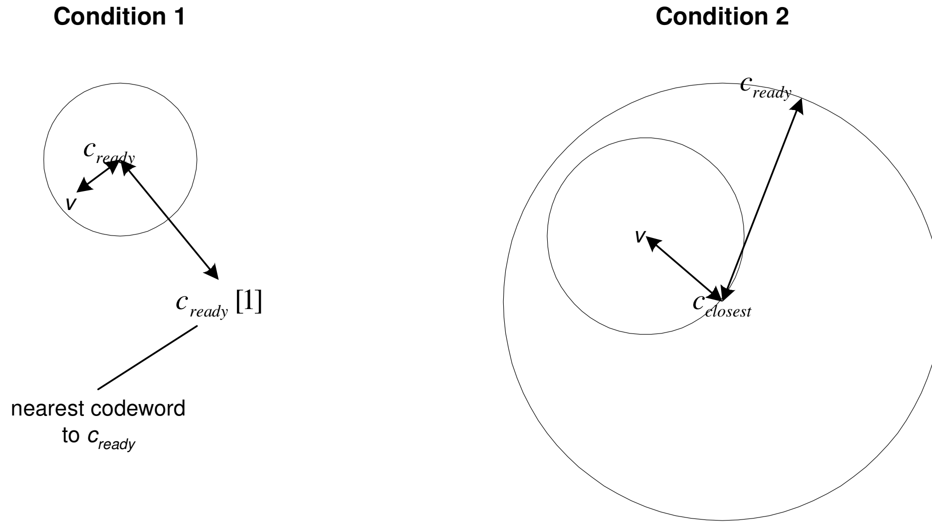

Figure 7: FE-UC: two termination conditions in the process of searching the closest codeword.

We use two methods for finding the initial codeword *s*. One is first compute the means of each codeword and the input vector *v*, and then scalar quantize the *v* to the closest codeword *s* according to the mean. The other is finding the closest codeword in the neighbours of *v*, which are already encoded (see Fig. 8). We use these codewords associated with neighbours as a prediction of initial codeword *s*, and find *s* among these codewords which has the closest distance to *v*. Usually we encode an image in a raster-scan order, therefore, most of the vectors have four neighbours already encoded and at least one neighbour for the vectors at the corners and sides, except the only one vector at the left-top corner. For this vector, we use Method One to obtain initial codeword s; while using Method Two for all the other vectors.

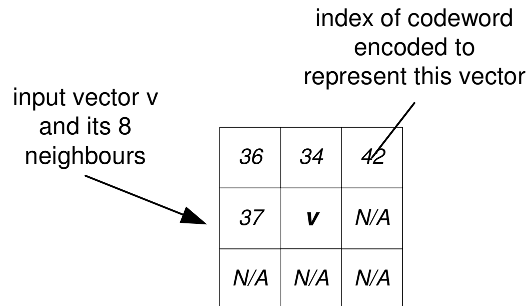

Figure 8: FE-UC: Method Two of obtaining the initial codeword.

3.2 Results

We use a general codebook with 256 entries and 16 dimensions of each codeword to test its encoding efficiency on three test images. The results are listed in Table VI.

Table VI. The average number of codewords compared when encoding an input vector

	Full Search Method	**FE-UC Method**
'Lena' 512×512×8 bits	256	8.04
'Pepper' 512×512×8 bits	256	10.35
'Goldhill' 512×512×8 bits	256	12.35

4. References

[Buzo, 1980] A. Buzo, A. H. Gray, R. M. Gray, and J. D. Markel, "Speech Coding Based upon Vector Quantization", IEEE Trans. Acoust. Speech & Signal Processing, 28, 562-574, 1980.

[Cockshott, 1999] W.P. Cockshott, R. Lambert, "Algorithm for Hierarchical Vector Quantization of Video Data", IEE Proc.-Vis. Image Signal Process, 146(4), 222-228, Aug. 1999.

[Equitz, 1989] W. H. Equitz, "A new vector quantization clustering algorithm", IEEE Trans. Acoust., Speech, Signal Process., vol. 37, pp. 1568-1575, Oct. 1989.

[Gray, 1998] R.M. Gray, D.L. Neuhoff, "Quantization", IEEE Trans. Inform. 44(6), 2325-2383, Oct., 1998.

[Linde, 1980] Y. Linde, A. Buzo, R. M. Gray, "An Algorithm for Vector Quantizer Design", IEEE Trans. Commun. 28, 84-95, Jan., 1980.

[Lloyd, 1982] S. P. Lloyd, "Least Squares Quantization in PCM", IEEE Trans. Inform. Theory, 28, 129-137, Mar. 1982.

[Shannon, 1959] C.E. Shannon, "Coding Theorems for a Discrete Source with a Fidelity Criterion", IRE Nat. Conv. Rec., part 4, 142-163, 1959.

[SVD, Web] http://mathworld.wolfram.com/SingularValueDecomposition.html