General-Tree-Structured Vector Quantizer for Image Progressive Coding Using the Smooth Side-Match Method

Shiueng-Bien Yang

***Abstract*—Several tree-structured vector quantizers have recently been proposed. However, owing to the fact that all trees used are fixed M-ary tree-structured, the training samples contained in each node must be artificially divided into a fixed number of clusters. This paper presents a general-tree-structured vector quantizer (GTSVQ) based on a genetic clustering algorithm that can divide the training samples contained in each node into more natural clusters. Also, the Huffman tree decoder is used to achieve the optimal bit rate after the construction of the general-tree-structured encoder. Progressive coding can be accomplished by giving a series of distortion or rate thresholds. Moreover, a smooth side-match method is presented herein to enhance the performance of coding quality according to the smoothness of the gray levels between neighboring blocks. The combination of the Huffman tree decoder and the smooth side-match method is proposed herein. Furthermore, the Lena image can be coded by GTSVQ with 0.198 bpp and 34.3 dB in peak signal-to-noise ratio.**

***Index Terms*—Genetic clustering algorithm, Huffman coding, smooth side-match method, tree-structured vector quantizer.**

I. Introduction

VECTOR QUANTIZATION (VQ) is a useful technique for data compression and coding, particularly in image and speech coding. The design of the codebook is the key issue for vector quantization. Two points are important in codebook design. First, a good set of codewords that minimizes quantization errors must be found. The LBG algorithm [1] had been widely used in codebook design. However, the LBG algorithm also suffers another drawback in that the user must provide the number of clusters in advance, yet generally has no idea about how many clusters there should be in the data set. This paper uses a genetic clustering algorithm ([2], [3]) in the codebook design. This algorithm searches for an appropriate number of cluster centers and then simultaneously performs the clustering. The user need not be concerned about the number of clusters in the genetic clustering algorithm. The second important point is to try to make the codebook search as fast as possible. The full search VQ searches the whole codebook sequentially, and requires a time of $O(n)$ when the codebook contains n codewords. In the tree-structured vector quantizers (TSVQ) ([4]–[10]), the codebook search requires a time of about $O(\log n)$ when the tree-structured codebook is roughly a balanced tree. The above marks a great improvement in time complexity because a codebook may be large.

In [4], the TSVQ was designed one layer at a time using the generalized Lloyd algorithm. Each new layer of the tree was obtained by splitting each leaf node of the previous layer into two nodes, and thus the tree grows into a balanced tree that implements a fixed-rate code. In [5], an alternative TSVQ design algorithm was introduced. The tree is grown one node at a time rather than one layer at a time, and the node that contributes most to decreasing the overall distortion is chosen for splitting. The tree is grown into an unbalanced tree whose leaves equal a predetermined number, which is a power of two. Chou *et al.* [6] proposed a method for designing an unbalanced tree coder. First, a balanced fixed-rate TSVQ is grown to a predetermined height, and is then optimally pruned back by using the generalized Breiman, Friedman, Olshen, and Stone (BFOS) algorithm [11]. This algorithm works by always pruning off the subtree with the smallest value of λ, where λ is defined as the ratio of the increase in distortion to the decrease in rate. Riskin *et al.* [7] proposed a greedy method for node splitting and tree growing. Balakrishnan *et al.* [8] proposed a recursive splitting process to grow the tree, which outperforms [7]. All of the tree-structured coders mentioned above are fixed M-ary trees, meaning that when a node is split, the training samples contained in that node have to divide into a fixed number of clusters. However, cases usually occur where it is inappropriate to divide the set of training samples into a fixed number of clusters. In this paper, the genetic clustering algorithm is presented to search for a proper number of clusters in the data set by itself. Applying this genetic clustering algorithm allows a general-tree-structured vector quantizer to be proposed. A distortion or rate threshold is used as a stopping threshold of growing the general-tree coder. Furthermore, all leave nodes in the general-tree coder are used to build a Huffman tree [12] decoder. Specifying a series of distortion or rate thresholds also allows the implementation of the progressive coding.

Side-match vector quantizer (SMVQ) [13] is a well-known class of finite-state vector quantizers (FSVQ) and several variants have been proposed (e.g., [14]–[16]). SMVQ selects the codewords used to construct the state codebook such that the gray levels of pixels across the boundaries of neighboring blocks are as uniform as possible. However, in real images, the change of gray levels among the neighboring pixels is generally smooth. In this paper, the smooth side-match method is presented. The smooth side-match method selects the codeword nearest to the encoded block according to the smoothness of the gray levels of pixels between neighboring blocks. In GTSVQ, we use the smooth side-match method to select the better codeword in the decoder, thus enhancing the coding quality.

The remainder of this paper is organized as follows. Section II presents the design of the smooth side-match method, while Section III describes the designs of the general-tree coder

Manuscript received October 4, 2000; revised October 2, 2002. This work was supported by the National Science Council of the Republic of China under Contract NSC 90-2213-E-426-001. This paper was recommended by Associate Editor H. Sun.

The author is with the Department of Computer Science and Information Engineering, Leader University, Tainan City, Taiwan, R.O.C. (e-mail: ysb@mail.leader.edu.tw).

Digital Object Identifier 10.1109/TCSVT.2002.808444

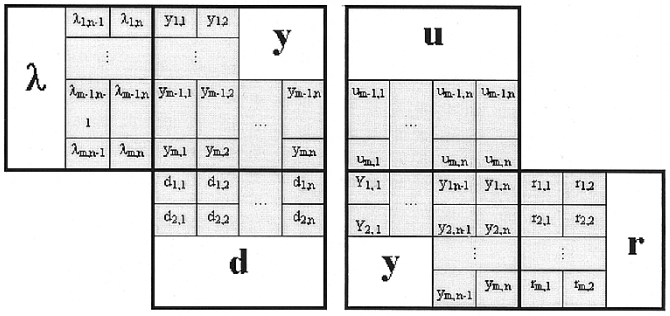

Fig. 1. Neighboring blocks used in the smooth side-match method.

and the Huffman tree decoder for GTSVQ. Section IV provides the design of the GTSVQ, while Section V describes the genetic clustering algorithm. Finally, experiments are given in Section VI and conclusions are presented in Section VII.

II. Design of Smooth Side-Match Method

SMVQ tries to make the gray levels of pixels right across the boundaries of neighboring blocks as near as possible. However, if the gray levels of pixels across the boundaries between neighboring blocks is increasing or decreasing, SMVQ does not encode a block well. In real images, the changing of gray levels is in general smooth among the neighboring pixels. We use this property to define the smooth side-match distortion. We first define the difference $\mathbf{dif}(e, f)$ between the pixels e and f as follows:

$$\text{dif}(e, f) = (\text{gray level of } e) - (\text{gray level of } f). \quad (1)$$

In GTSVQ, the diagonal blocks are selected as the basic blocks, and they are encoded first. The encoded image is therefore divided into two parts, the upper triangular region and the lower triangular region. In the upper triangular region, the neighboring blocks of the currently encoded block are defined to be its left block and its lower block. An example is shown in Fig. 1(a). The vertical smooth side-match distortion is defined as

$$\text{Upper\_}vd(y) = \sum_{i=1}^{n} \left| \frac{(\text{dif}(d_{2,i}, d_{1,i}) + \text{dif}(y_{m,i}, y_{m-1,i}))}{2} - \text{dif}(d_{1,i}, \text{y}_{m,i}) \right| \quad (2)$$

and the horizontal smooth side-match distortion is defined as

$$\text{Upper\_}hd(y) = \sum_{j=1}^{m} \left| \frac{(\text{dif}(\ell_{j,n-1}, \ell_{j,n}) + \text{dif}(y_{j,1}, y_{j2}))}{2} - \text{dif}(\ell_{j,n}, y_{j,1}) \right|. \quad (3)$$

Then, the smooth side-match distortion of the codeword y is defined as

$$D(y) = \text{Upper\_}vd(y) + \text{Upper}hd(y). \quad (4)$$

In the lower triangular region, the neighboring blocks of each encoded block are defined to be its right block and its upper block. An example is depicted in Fig. 1(b). The vertical smooth side-match distortion is defined as

$$\text{Lower\_}vd(y) = \sum_{i=1}^{n} \left| \frac{(\text{dif}(u_{m-1,i}, u_{m,i}) + \text{dif}(y_{1,i}, y_{2,i}))}{2} - \text{dif}(u_{m,i}, y_{1,i}) \right| \quad (5)$$

and the horizontal smooth side-match distortion is defined as

$$\text{Lower\_}hd(y) = \sum_{j=1}^{m} \left| \frac{(\text{dif}(r_{j,2}, r_{j,i}) + \text{dif}(y_{j,n}, y_{j,n-1}))}{2} - \text{dif}(r_{j,1}, y_{j,n}) \right| .quad (6)$$

Then, the smooth side-match distortion of the codeword y is defined as

$$D(y) = \text{Lower\_}vd(y) + \text{Lower\_}hd(y). \quad (7)$$

In GTSVQ, the codeword that has the smallest $D(y)$ is regarded as the decoding result.

III. Designs of the General-Tree Coder and the Huffman-Tree Decoder

Before describing how the general-tree coder and the Huffman-tree decoder are constructed, the data fields used are introduced. The structure of data fields for a node in the general-tree coder is (distortion level, codeword, code, number of training samples), and the structure of data fields for a node in the Huffman-tree decoder is (distortion level, codeword, number of training samples). The "distortion level" field is used mainly for progressive coding, in which many coding levels are required to encode an image. For each coding level, the coder may send the partial code of input data to the decoder, and the input data can then be decoded by this partial code in the decoder. After the final coding level is finished, input data can be completely decoded in the decoder. The "distortion level" records whether or not a node in the tree can be used to encode input data in this coding level during progressive coding. The "codeword" fields of leaf nodes are the codewords of this tree-structured codebook. The "code" field of each leaf node contains the code for the corresponding codeword. The "number of training samples" field contains the size of the set of training samples.

Section III-A describes the design of the general-tree coder and the Huffman tree decoder with the distortion thresholds, while Section III-B presents the design of the general-tree coder and the Huffman tree decoder with the rate thresholds.

A. Distortion Thresholds

The average distortion of node X is first defined as follows. Let X be a node containing a set of training samples $\{O_1, O_2, \ldots, O_n\}$ and let S be the center of this set. Notably, the "center of the set" indicates the mean value of the samples in this set. The average distortion of node X is defined as

$$D(X) = \frac{\sum_{j=1}^{n} ||O_j - S||}{n} \quad (8)$$

where $||O_j - S||$ is the Euclidean distance between vectors O_j and S.

The algorithm CLUSTERING, described in Section V, is used to design the general-tree coder with distortion thresholds as follows. At first, the first stage of CLUSTERING is applied to the set of whole training samples and a set of connected components $\{B_1, B_2, \ldots, B_m\}$ is obtained. The distortion threshold is defined as follows:

$$\epsilon = v^* \max_{1 \le i \le m} D(B_i),$$
$$1 \le v \le \frac{D(\text{the set of whole training samples})}{\max_{1 \le i \le m} D(B_i)} \quad (9)$$

where D (the set of whole training samples) is defined as (8) in this section, and where v is given by the user. After the tree coder had been constructed, the average distortion of each leaf node will be less than or equal to ε, implying that the average quantization error will be less than or equal to ε when the tree coder is used to code all the training samples.

The genetic algorithm is then applied to $\{B_1, B_2, \ldots, B_m\}$ in the second stage of CLUSTERING. The details of the general-tree coder design process are described in the following.

Algorithm:
General_Tree_Construction_Distortion_Thresholds

Input: A set of training objects $\{B_1, B_2, \ldots, B_m\}$ contained in a node B, where each B_i is itself a set of objects. The distortion threshold ϵ and the coding level c.

Output: A general-tree coder T_B, with an average distortion for each leaf node of less than or equal to ϵ, and with the Huffman decoding tree H_B.

Step 1) Set node B to the root node of the general-tree coder T_B. Apply the second stage of CLUSTERING to the set of training samples contained in the node B, obtaining p clusters $C_1, C_2, \ldots, C_p$. Each cluster C_i is designed as a child node of node B in T_B.

Step 2) For each leaf node Y in the general-tree coder T_B, do the following.

Step 2.1)

If The average distortion of node Y is greater than ϵ.

Then Set the distortion level of node Y to 0. Apply the second stage of CLUSTERING to the set of training samples contained in the node Y to design the child nodes of node Y in T_B.

Else Set the distortion level of node Y to c.

Step 3) (The coder tree T_B has been constructed). Take all the leaf nodes in T_B to design the corresponding Huffman tree decoder H_B based on the values of the "number of training samples" field.

Step 4) Traverse each leaf node of H_B to find the code for this node and put this code into the "code" field of the corresponding node in the coder tree T_B. The "distortion level" field of internal nodes are set to 0, and $c$for the leaf nodes in T_B and H_B. Stop.

The construction of the general-tree coder for progressive coding, when k distortion thresholds are set as $\epsilon_1 > \epsilon_2 > \ldots > \epsilon_k$, is described as follows. Initially, the algorithm General_Tree_Construction_Distortion_Thresholds is used to construct the coder T_1 with one coding level and the corresponding decoder H_1, with the value of coding level c set as 1. To make progressive coding possible, the construction of the coder tree T_k with k coding levels and the corresponding Huffman decoding tree H_k is described herein, given that the coder T_{k-1} and the decoder H_{k-1}. The construction algorithm is presented below.

Algorithm:
Progressive_General_Tree_Construction_Distortion_Thresholds

Input: A general-tree coder T_{k-1} with $k - 1$ coding levels and the corresponding Huffman decoding tree H_{k-1}. The distortion threshold ϵ_k and the coding level k.

Output: A general-tree coder T_k with average distortion of each leaf node less than or equaling to ϵ_k and the Huffman decoding tree H_k. T_k is capable of progressive coding with $k$coding levels.

Step 1) For each leaf node Xin T_{k-1} do the following.

Step 1.1) Let the node be X.

If The average distortion of X is greater than ϵ_k.

Then Apply the algorithm General_Tree_Construction_Distortion_Thresholds to the node X, given two parameters, the distortion threshold ϵ_k and the coding level k. The general-tree coder T_X and corresponding Huffman decoding tree H_X are then obtained. Let node X in T_{k-1} be the root node of T_X, and let the corresponding node X in H_{k-1} be the root node of H_X.

Else Set both distortion levels of node X in T_{k-1} and H_{k-1} to k.

Step 2) Stop.

B. Rate Thresholds

This section designs the growing method for the general-tree coder before describing the designs of the general-tree coder and the Huffman tree decoder with the rate thresholds. In our approach, splitting one node at a time grows the general-tree. Let S denote the set of leave nodes in the general-tree R, including

node X. Let $P(X)$ represent the probability on the training samples in node X, and let $R(X)$ denote the bits required to represent node X. They are defined as

$$P(X) = \frac{\text{the number of training samples contained in node X}}{\text{all of the training samples}} \tag{10}$$

$$R(X) = \text{the number of bits representing the Huffman code of the node X.} \tag{11}$$

Finally, let $D(T)$ and $R(T)$ indicate the distortion and rate, respectively, measured by T. The tree T' then indicates the tree T after node X is split into nodes, $X_1, X_2, \ldots, X_p$, by the second stage of CLUSTERING. Then

$$D(T) = \sum_{\substack{i \in S \\ i \neq X}} P(i)D(i) + P(X)D(X) \tag{12}$$

$$R(T) = \sum_{\substack{i \in S \\ i \neq X}} P(i)R(i) + P(X)R(X) \tag{13}$$

$$D(T') = \sum_{\substack{i \in S \\ i \neq X}} P(i)D(i) + \sum_{i=1}^{p} P(X_i)D(X_i) \tag{14}$$

$$R(T') = \sum_{\substack{i \in S \\ i \neq X}} P(i)R(i) + \sum_{i=1}^{p} P(X_i)R(X_i). \tag{15}$$

Thus, the ratio of the change in distortion to change in rate due to splitting leaf node X, is defined as

$$\lambda = -\frac{D(T') - D(T)}{R(T') - R(T)} = -\frac{P(X)D(X) - \sum_{i=1}^{p} P(X_i)D(X_i)}{P(X)R(X) - \sum_{i=1}^{p} P(X_i)R(X_i)} \tag{16}$$

The design of the growing method is to select the node with the largest λ to be split at a time in general-tree coder T. The process of growing algorithm `Find_Split_Node` is detailed as follows.

Algorithm: Find_Split_Node
Input: A general-tree coder T.
Output: A splitting node $\hat{X}$ in the coder tree T and the value of λ for node $\hat{X}$.

Step 1) Set queues Q and Q' to be empty.

If Only one node, namely the root node, exists in the coder tree T.
Then Put the root node to the queue Q.
Else Put all of the leaf nodes of coder tree T to the queue Q.

Step 2) For each node X in queue Q, do the following:
- Step 2.1) Apply the second stage of CLUSTERING to the set of training samples contained in node X, thus obtaining p clusters $C_1, C_2, \ldots, C_p$.
- Step 2.2) Design each child node X_i of node X, for $1 \leq i \leq p$, but do not split node X in coder tree T. For each node X_i, set the pointer of node X_i to C_i, fill the size of C_i to the field "number of training samples" and compute the center of C_i.
- Step 2.3) Let T' denote the coder tree after splitting node X in T. Use these leaf nodes in T' to design a Huffman decoding tree H' based on the values of the fields "number of training samples". Traverse each leaf node of H' to find the code for this node and put this code into the "code" field of the corresponding node in coder tree T'.
- Step 2.4) Calculate the value of λ as in (16) when node X in the coder tree T is split to form coder tree T'. Insert the value of λ to the queue Q'.

Step 3) Search queue Q' for the maximum value of λ, and let node $\hat{X}$ have a maximum value of λ in Q'. The node $\hat{X}$ and its corresponding λ are the output.
Step 4) Stop.

Designs of the general-tree coder and the Huffman tree decoder with the rate thresholds are described as follows. The first stage of CLUSTERING is applied to the set of whole training samples, thus obtaining a set of connected components $\{B_1, B_2, \ldots, B_m\}$. Then, the second stage of CLUSTERING is applied to $\{B_1, B_2, \ldots, B_m\}$ to construct the general-tree coder and the Huffman tree decoder with the rate thresholds. Before designing the general-tree coder, the user must give a rate threshold for the general-tree coder. The general-tree is grown one node at a time. The growing process continues until the average rate of the general-tree reaches the rate threshold. The algorithm for designing the general-tree coder and the Huffman-tree decoder with rate thresholds is as follows.

Algorithm:
General_Tree_Construction_Rate_Thresholds
Input: A set of training objects $\{B_1, B_2, \ldots, B_m\}$ is contained in a node B, where each B_i is itself a set of objects. The rate threshold R_c and the coding level c.
Output: A general-tree coder T_B with an average rate of less than or equal to R_c and the Huffman tree decoder H_B.

Step 1) Set node B as the root node of tree T_B. Set Rate $= 0$.
Step 2) **while** The value of Rate is smaller than R_c.
- Step 2.1) Apply the algorithm `Find_Split_Node` to the tree T_B. Let node $\hat{X}$ denote the node to be split in T_B, then split node $\hat{X}$ in the coder tree T_B to construct a new coder tree T'_B. Use all the leaf nodes of the general-tree coder T'_B to construct the corresponding Huffman tree decoder H'_B. Traverse each leaf node of H'_B to find the code and put this

code into the "code" field of the corresponding node in the coder tree T'_B.

Step 2.2) Rate $= R(T'_B)$, where $R(T'_B)$ is defined as in (15). Set $T_B = T'_B$ and $H_B = H'_B$.

Step 3) Apply the algorithm `Find_Split_Node` to the tree T_B. Let node $\hat{X}$ denote the node to be split in T_B, then split node $\hat{X}$ in the coder tree T_B to construct a new coder tree T'_B. Use all the leaf nodes of the general-tree coder T'_B to construct the corresponding Huffman tree decoder H'_B. Traverse each leaf node of H'_B to find the code and put this code into the "code" field of the corresponding node in the coder tree T'_B.

Step 4) Rate $= R(T'_B)$, where $R(T'_B)$ is defined as in (15). Set $T_B = T'_B$ and $H_B = H'_B$.

Step 5) Apply the algorithm `Find_Split_Node` to the tree T_B. Let node $\hat{X}$ denote the node to be split in T_B, then split node $\hat{X}$ in the coder tree T_B to construct a new coder tree T'_B. Use all the leaf nodes of the general-tree coder T'_B to construct the corresponding Huffman tree decoder H'_B. Traverse each leaf node of H'_B to find the code and put this code into the "code" field of the corresponding node in the coder tree T'_B.

Step 6) Rate $= R(T'_B)$, where $R(T'_B)$ is defined as in (15). Set $T_B = T'_B$ and $H_B = H'_B$.

Step 7) Set the distortion levels at 0 and c for the internal and leaf nodes in T_B, respectively.

Step 8) Output coder tree T_B and corresponding decoder tree H_B. Stop.

This paper describes how to construct the general-tree coder for progressive coding, when k rate thresholds are set as $R_1 < R_2 < \ldots < R_k$, as follows. First, the algorithm `General_Tree_Construction_Rate_Thresholds` is used to construct the coder T_1 with one coding level and the corresponding decoder H_1, given a coding level c of 1 and a rate threshold of R_1.The construction of the coder tree T_k with $k$coding levels and the corresponding Huffman decoding tree H_k is then described, given that the coder T_{k-1} and the decoder H_{k-1}. The construction algorithm is presented below.

Algorithm:
Progressive_General_Tree_Construction_Rate_Thresholds

Input: A general-tree coder T_{k-1} with k-1 coding levels and the corresponding Huffman decoding tree H_{k-1}. The rate threshold R_k and the coding level k.

Output: A general-tree coder T_k with an average rate of less than or equal to R_k and the Huffman decoding tree H_k. T_k is capable of progressive coding with k coding levels.

Step 1) Let there be q leave nodes, $X_1, X_2, \ldots, X_q$, in T_{k-1}. Set each node X_i to be the root node of tree T_{X_i}, for $1 \le i \le q$. Set Rate $= R(T_{k-1})$ and $R(T_{X_i}) = 0$, for $1 \le i \le q$.

Step 2) **while** The value of Rate is smaller than R_k.

Step 2.1) Set the queue Q' to be empty.

Step 2.2) For each tree, T_{X_i}, for $1 \le i \le q$, do the following.

Step 2.2.1) Apply the algorithm Find_Split_Node to the tree T_{X_i}. Let node X' be the node to be split in T_{X_i}. Insert the corresponding value of λ for node X' into queue Q'.

Step 2.3) Search queue Q' for the maximum value of λ. Let the node $\hat{X}$ have a maximum value of λ in Q', and let node $\hat{X}$ be contained in tree T_{X_k}. Split node $\hat{X}$ in the coder tree T_{X_k} to construct a new coder tree T'_{X_k}. All the leaf nodes of coder tree T'_{X_k} are used to construct the corresponding Huffman decoder tree H'_{X_k}.

Step 2.4) Let ΔR be the increased rate when the node $\hat{X}$ is split in coder tree T_{X_k} to form the coder tree T'_{X_k}. Set $\Delta R = R(T'_{X_k}) - R(T_{X_k})$.

Step 2.5) Rate $=$ Rate $+ \Delta R$. Set $T_{X_k} = T'_{X_k}$ and $H_{X_k} = H'_{X_k}$.

Step 3) Set the distortion levels at 0 and k for the internal and leaf nodes in T_{X_i}, respectively, for $1 \le i \le q$.

Step 4) Set each leaf node X_i in T_{k-1} as the root node of T_{X_i} and set the corresponding node X_i in H_{k-1} as the root node of H_{X_i}, for $1 \le i \le q$.

Step 5) Output the coder tree T_k and the corresponding decoder tree H_k. Stop.

IV. Design of the GTSVQ

This section describes the design of GTSVQ. [17]–[19] used discrete cosine transform (DCT) coefficients as the edge oriented features. However, in GTSVQ, all training images were divided into blocks of size 4×4, and each block was then transformed into the DCT coefficients by DCT. Six DCT coefficients were sufficient to serve as clustering features, where c(0, i)'s represent the horizontal features and c(i, 0)'s denote the vertical features for $1 \le i \le 3$. The CLUSTERING algorithm is used to construct the general-tree coder and corresponding Huffman tree decoder. However, the codeword contained in each leaf node in the Huffman tree decoder takes the form of the DCT coefficients. To make it possible to apply the smooth-side match method in the decoding phase, the inversed DCT (IDCT) was used to transform the DCT coefficients in each leaf node into a spatial vector, with this spatial vector representing the codewords. The smooth side-match distortions of the codewords can then be calculated in the decoder, and the codeword with the smallest smooth side-match distortion is the decoding result. Fig. 2 presents an example illustrating the encoder and the decoder. The encoder tree T_3 is employed in Fig. 2 to perform progressive coding with three coding levels, meaning three codes are obtained when a block is encoded, one for each coding level.

The image coding process for GTSVQ will be described next. Initially, all the diagonal blocks, those from the left-upper block to the right-lower block in an image, are encoded directly using

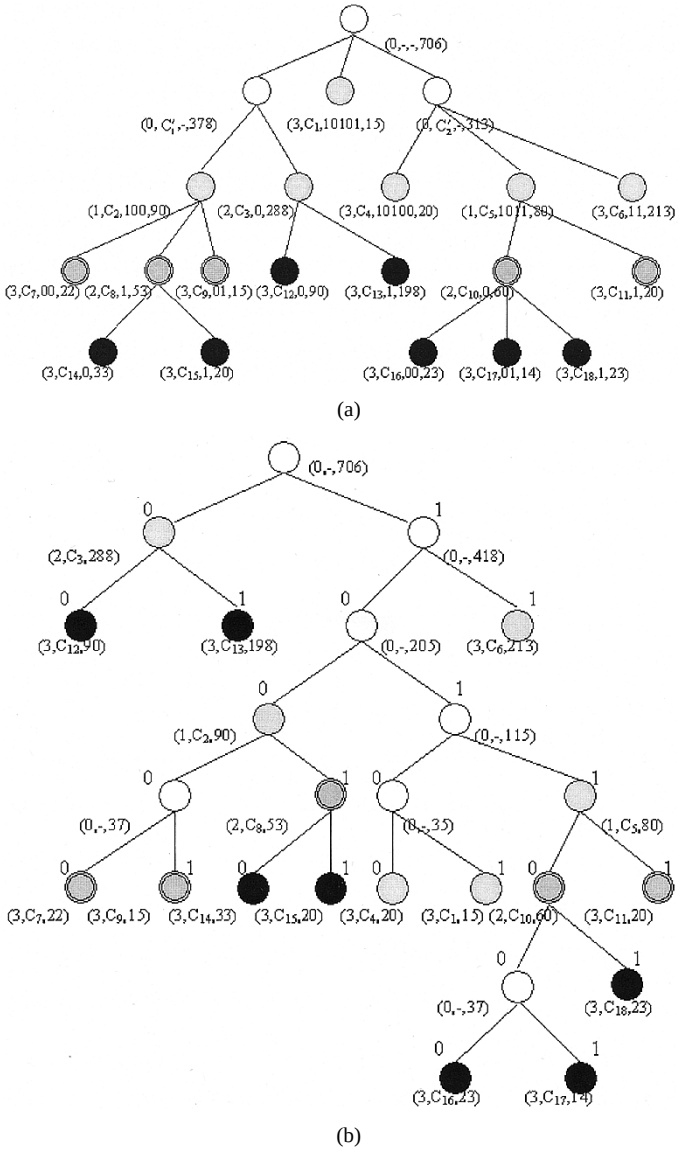

Fig. 2. Example of the coder tree T_3 and the Huffman decoding tree H_3. (a) The coder tree T_3; (b) The Huffman decoding tree H_3.

all of the codewords contained in the leave nodes of the general-tree coder T_3, and are then decoded using the Huffman tree decoder H_3. After the diagonal blocks are coded, the other blocks are progressively encoded. Meanwhile, coding a vector qusing the coder tree T_3 with three coding levels involves the following steps. In the first coding level, the distances $\|q-C'_1\|$, $\|q-C_1\|$, and $\|q-C'_2\|$ are computed. Suppose q is nearest to C'_2, check the value of the distortion level. If the distortion level equals zero, further steps are required. Then, the distances $\|q-C_4\|$, $\|q-C_5\|$, and $\|q-C_6\|$ are calculated. Suppose $q$is closest to C_5, since the distortion level of this node C_5 is 1, the process is stopped and the code 1011 contained in C_5 is sent to the Huffman decoding tree. In the Huffman tree decoder, two steps are required to determine the decoding result. First, the decoder tree H_3 is traversed according to code 1011 and then node C_5 is obtained in H_3. Second, the smooth side-match distortions are calculated for all of the leaf nodes contained in the subtree rooted at node C_5, namely C_{11}, C_{16}, C_{17}, and C_{18}. The codeword with the smallest smooth side-match distortion among C_{11}, C_{16}, C_{17}, and C_{18} is the decoding result. The second coding level requires further steps, because the coding level of this node C_5 (which is 2) is greater than the distortion level (which is 1). Distances $\|q-C_{10}\|$ and $\|q-C_{11}\|$ are computed. Suppose q is nearer to C_{10}; then, since the distortion level of this node is 2, the process stops, and code 0 is sent to the Huffman decoding tree. Similarly, node C_{10} is obtained when code 0 is received by decoder H_3. Also, the smooth side-match distortions for all of the leaf nodes contained in the subtree rooted at node C_{10} are calculated, namely C_{16}, C_{17} and C_{18}, and that with the smallest smooth side-match distortion is the decoding result in the second coding level. Furthermore, in the third coding level, because the coding level of this node C_{10} (which is 3) is greater than the distortion level (which is 2), further steps are required. Consequently, the distances $\|q-C_{16}\|$, $\|q-C_{17}\|$, and $\|q-C_{18}\|$ must be calculated. Suppose q is nearest to C_{17}, the code 01 contained in node C_{17} is sent to the decoder H_3. Because the coding level (which is 3) is now no greater than the distortion level of node C_{17} (which is 3), the coding process is stopped. Finally, the codeword C_{17} is obtained in H_3. Clearly, the smooth side-match method need not be used at the third coding level.

However, the distortion level of node C_3 in T_3 is 2, meaning that if node C_3 is used to encode some block in the first coding level, it can also be used to encode this block in the second coding level. Thus, the coder does not send any code for this block to the decoder in the second coding level until the third coding level begins. In the second coding level, the decoder H_3 only needs to calculate the smooth side-match distortions of two nodes C_{12} and C_{13} and outputs the better one as the decoding result.

Notably, in the case of a single coding level, the Huffman tree decoder achieves the optimal Huffman codes. In the progressive coding case, the Huffman tree decoder does not achieve the optimal Huffman codes, since the Huffman tree decoder is the combination of several Huffman trees. That is, the partial codes sent from the coder to the decoder for each coding level are optimal Huffman codes. However, the total of bit rate received in the decoder is clearly not optimal.

V. The Genetic Clustering Algorithm

The CLUSTERING algorithm uses the genetic strategy to automatically search for the proper number of clusters and find a good clustering. The algorithm CLUSTERING consists of two stages. The first stage is the nearest neighbor (NN) algorithm [20]. The distance used to group objects in the NN algorithm is based on the average of the nearest neighbor distances. Meanwhile, the second stage consists of a heuristic method and a genetic algorithm. The heuristic method is used to identify a good clustering by applying the genetic algorithm [21].

The design of the codebook denotes each training block with a size of 4×4 as an object in the CLUSTERING algorithm. Let there be n objects, $O_1, O_2, \ldots, O_n$, and suppose that each object is characterized by p feature values, then $O_i = (o_{i1}, o_{i2}, \ldots, o_{ip}) \in R^p$. The first stage of CLUSTERING, that is, the NN algorithm, is described below.

Step 1) For each object O_i, find the distance between O_i and its nearest neighbor. That is

$$d_{\mathrm{NN}}(O_i) = \min_{j \neq i} ||O_j - O_i||, \text{where} ||O_j - O_i|| = \left(\sum_{q=1}^{p} (O_{jq} - O_{iq})^2 \right)^{\frac{1}{2}}. \quad (17)$$

Step 2) Compute d_{av}, the average of the nearest neighbor distances, as follows:

$$d_{\mathrm{av}} = \frac{1}{n} \sum_{i=1}^{n} d_{\mathrm{NN}}(O_i). \text{Let d} = u d_{\mathrm{av}}. \quad (18)$$

Step 3) View the n objects as nodes of a graph, and compute the adjacency matrix $A_{n \times n}$ as follows:

$$A(i,j) = \begin{cases} 1, & \text{if} ||\mathrm{O}_i - \mathrm{O}_j|| \leq \mathrm{d} \\ 0, & \text{otherwise} \end{cases} \quad (19)$$

where $1 \leq j \leq i \leq n$.

Step 4) Find the connected components of this graph, and let them be denoted by $B_1, B_2, \ldots, B_m$.

[21] showed that the value of u may be chosen from the interval [1.4, 1.8], and that the exact value is not critical to the clustering result. Herein, u is set at 1.5. For the genetic algorithm, the initial data set contains m small sets $B_1, B_2, \ldots, B_m$, where the center of each set B_i is denoted by V_i for $1 \leq i \leq m$. Clearly, m is smaller than the original data set, which is n. The objective of using the NN algorithm in the first stage is to reduce computational time and space during the second stage. Hence, the genetic algorithm in the second stage can process large data sets efficiently.

The genetic algorithm consists of an initialization step and the iterative generations with three phases, namely the reproduction, crossover and mutation phases. They are described as follows.

Initialization Step

Let N be the population size in the genetic algorithm. A population of N binary strings is randomly generated. The length of each string is m, which is the number of sets obtained in the first stage. N binary strings are generated in such a way that the number of 1's in the strings is almost uniformly distributed within [1,m]. Meanwhile, each string represents a subset of $\{B_1, B_2, \ldots, B_m\}$. If B_i is in this subset, the ith position of the string will be 1; otherwise, it will be 0. Each B_i in the subset is used as a seed to generate a cluster.

Before describing the three phases, the method of generating a clustering from the seeds is described. Let $T = \{T_1, T_2, \ldots, T_s\}$ be the subset corresponding to a string, initial clusters C_i be T_i for $i = 1, 2, \ldots, s$. Furthermore, let the initial centers S_i of clusters C_i be V_i for $i = 1, 2, \ldots, s$ and let the size of cluster C_i be defined as $|C_i| = |T_i|$ for $i = 1, 2, \ldots, s$, where $|T_i|$ denotes the number of objects belonging to the set T_i.

Clusters are generated as follows. The B_i's in $\{B1, B_2, \ldots, B_m\} - T$ are taken one by one and the distance between the center of the taken B_i and each center S_j is calculated. Then we have

$$B_i \subset C_j, \text{ if } ||V_i - S_j|| \leq ||V_i - S_k||, \quad \text{for} \quad 1 \leq k \leq s. \quad (20)$$

If B_i is included in cluster C_j, the center S_j and the size of cluster C_j are recomputed as follows when B_i is added to C_j

$$S_j' = \frac{S_j |C_j| + V_i |B_i|}{|C_j| + |B_i|}, \quad |C_j'| = |C_j| + |B_i|. \quad (21)$$

After the B_i's in $\{B_1, B_2, \ldots, B_m\} - T$ have all been considered, the cluster C_j with center S_j generated by the seed T_j for $j = 1, 2, \ldots, s$ is obtained. $\{C_1, C_2, \ldots, C_s\}$ is defined as the set of clusters generated by this string.

Reproduction Phase

Let C_i be one of the clusters generated by string R. The following defines D_{intra} as the intra-distance in the cluster C_i and defines D_{inter} as the inter-distance between C_i and the set of all other clusters

$$D_{\mathrm{intra}}(C_i) = \sum_{B_k \subset C_i} ||V_k - S_i|| |B_k| \quad (22)$$

$$D_{\mathrm{inter}}(C_i) = \sum_{B_k \subset C_i} \left(\min_{i \neq j} ||V_k - S_j|| \right) |B_k| \quad (23)$$

where the summation is over all B_k's that are in cluster C_i. The fitness function of a string Rcan then be defined as follows:

$$\text{Fitness}(R) = \sum D_{\mathrm{inter}}(C_i) w - D_{\mathrm{intra}}(C_i) \quad (24)$$

where the summation is over all clusters C_i generated by string R and w denotes a weighting factor. If w is assigned a small value, the importance of $D_{\mathrm{intra}}(C_i)$ is emphasized, which tends to produce more numerous and compact clusters. Meanwhile, if w is assigned a large value, the importance of $D_{\mathrm{intra}}(C_i)$ is emphasized, which tends to produce fewer and looser clusters than if w is small. After the calculation of the fitness of each string in the population, the reproduction operator is implemented using a roulette wheel with slots sized according to fitness.

In the codebook design, the center of each cluster generated by the string with the highest fitness represents the codeword, since the string with the highest fitness represents a better clustering in the set of training objects.

Crossover Phase

If a pair of strings R and Q are chosen for applying the crossover operator, two random numbers p and q in $[1, m]$ are generated to decide which pieces of the strings are to be interchanged. Suppose $p < q$, then the bits from position p to position q of string R will be interchanged with those bits in the same positions of string Q. For each chosen pair of strings, the crossover operator is done with probability p_c. The significance of the crossover phase in codebook design lies in performing the interchange of the codewords contained in various strings.

Mutation Phase

During the mutation phase, the bits of the strings in the population are chosen with probability p_m. Each chosen bit is then

TABLE I
COMPARISON OF THE VARIOUS METHODS FOR THE CODED IMAGE "LENA"

Methods	Coding levels									
	(1)		(2)		(3)		(4)		(5)	
	PSNR (dB)	Rate (bpp)	PSNR (dB)	Rate (bpp)	PSNR (dB)	Rate (bpp)	PSNR (dB)	Rate (bpp)	PSNR (dB)	Rate (bpp)
VLTSVQ	24.8	0.263	26.0	0.322	27.9	0.380	30.2	0.445	32.9	0.518
VLTSVQ[Side]	29.3	0.264	30.8	0.323	31.7	0.382	32.5	0.446	32.9	0.519
VLTSVQ[Smooth]	30.0	0.264	31.3	0.323	32.2	0.382	32.6	0.446	32.9	0.519
PTSVQ	24.8	0.263	26.1	0.322	27.9	0.380	30.1	0.445	32.7	0.518
PTSVQ[Side]	29.1	0.264	30.5	0.323	31.5	0.382	32.3	0.446	32.7	0.519
PTSVQ[Smooth]	30.0	0.264	31.1	0.323	32.0	0.382	32.5	0.446	32.7	0.519
BTSVQ	24.5	0.268	25.8	0.325	27.7	0.382	29.9	0.448	32.5	0.521
BTSVQ[Side]	28.8	0.269	30.3	0.326	31.2	0.383	32.0	0.449	32.5	0.522
BTSVQ[Smooth]	29.5	0.269	30.7	0.326	31.6	0.383	32.2	0.449	32.5	0.522
VLTSVQ(M)	25.2	0.197	27.2	0.263	29.2	0.334	31.9	0.387	35.1	0.454
VLTSVQ(M)[Side]	31.6	0.198	32.9	0.264	33.9	0.335	34.7	0.388	35.1	0.455
VLTSVQ(M)[Smooth]	32.8	0.198	33.7	0.264	34.3	0.335	34.8	0.388	35.1	0.455
PTSVQ(M)	25.1	0.197	27.0	0.263	29.1	0.334	31.8	0.387	35.1	0.454
PTSVQ(M)[Side]	31.5	0.198	32.8	0.264	33.7	0.335	34.6	0.388	35.1	0.455
PTSVQ(M)[Smooth]	32.8	0.198	33.5	0.264	34.2	0.335	34.8	0.388	35.1	0.455
BTSVQ(M)	24.8	0.197	26.4	0.263	28.1	0.334	31.2	0.387	34.8	0.454
BTSVQ(M)[Side]	31.2	0.198	32.4	0.264	33.4	0.335	34.2	0.388	34.8	0.455
BTSVQ(M)[Smooth]	32.4	0.198	33.1	0.264	33.9	0.335	34.5	0.388	34.8	0.455
GTSVQ[VQ]	26.7	0.312	28.7	0.375	30.4	0.431	32.5	0.488	36.3	0.563
GTSVQ[LBG]	25.2	0.197	27.3	0.263	29.5	0.334	32.1	0.387	35.5	0.454
GTSVQ[Without]	25.8	0.197	28.5	0.263	31.1	0.334	33.2	0.387	36.1	0.454
GTSVQ[Side]	33.0	0.198	34.1	0.264	35.1	0.335	35.7	0.388	36.1	0.454
GTSVQ[Smooth]	34.3	0.198	34.9	0.264	35.5	0.335	35.8	0.388	36.1	0.454

changed from 0 to 1 or from 1 to 0, with a chosen cluster thus being discarded or produced in a string. Meanwhile, a codeword can be produced in codebook design by changing the corresponding bit from 0 to 1 or discarded by changing the corresponding bit from 1 to 0 during the mutation phase.

VI. EXPERIMENTS

The parameters used in the genetic algorithm in the experiments are as follows, namely a population size of 50, a crossover rate of 80%, and a mutation rate of 5%. One hundred generations were run and the best solution was retained. The parameter w had a value within $[3, 5]$, obtaining the number of branches between 2 and 20 when each node is split. Meanwhile, u was set to 1.5 in the first stage of CLUSTERING. In [3], it was shown that the values of u and w were not critical.

Five 512 $\times$ 512 (pixels) images with 256 gray levels were employed as the training images, and were divided into 4 $\times$ 4 blocks to construct the general-tree coder and corresponding Huffman tree decoder. Table I compares the proposed method with the other three methods. Three other methods, Balakrishnan's method [8], Chou's method [6], and the balanced binary-tree VQ were implemented for purposes of comparison with the proposed method. These three methods were termed VLTSVQ, PTSVQ, and BTSVQ. In PTSVQ, the tree was grown to a height of 11 before pruning. Also, these three TSVQs are easily designed using the M-ary trees, namely VLTSVQ(M), PTSVQ(M), and BTSVQ(M), where M denotes the average number of children of a node in the general-tree coder designed by GTSVQ. In Table I, M is set to five for these three TSVQ(M)s, and the Huffman coding is also used to design the codes in the TSVQ(M)s, as considered in GTSVQ. To compare the various methods listed in Table I, each coder is designed using the same structure. The CLUSTERING algorithm was first used to design the general-tree coder with five coding levels, and then these three TSVQs were applied separately to obtain the same size for each codebook by using the clustering algorithm LBG. The numbers of codewords for the five coding levels listed in Table I are 23, 58, 78, 171, and 302. GTSVQ[VQ] indicates the coding result by searching the whole leaf nodes in the general-tree coder. Meanwhile, GTSVQ[Without] indicates the coding result of GTSVQ without using the smooth side-match method in the decoder. Thus, the second step in the decoder could be omitted in GTSVQ[Without]. The GTSVQ[Side] and GTSVQ[Smooth] show the coding results of the GTSVQ using the conventional side-match method and the smooth side-match method, respectively, in the second step of the decoder. However, these three TSVQ's can use the smooth side-match method in the decoder just as the GTSVQ[Smooth]. For example, the coding quality of the "Lena" image encoded by VLTSVQ[Smooth] and VLTSVQ(M)[Smooth] in Table I is better than that encoded by VLTSVQ and VLTSVQ(M), respectively. Furthermore, the coding quality of GTSVQ[Smooth] is better than that of VLTSVQ(M)[Smooth] encoded, because the CLUSTERING algorithm can search the proper number of child nodes when splitting a node in the coder tree. Notably, in Table I, the coding quality of the "Lena" image encoded by GTSVQ[Smooth] is 34.3 dB at 0.198 bpp in the first coding level and 36.1 dB at 0.454 bpp in the fifth coding level. That is, the PSNR gain from doubling the coding rate at 0.198 bpp is within 2 dB. This phenomenon occurs because the codeword encoded for each input block in the first coding level is usually the same as the codeword encoded in the fifth coding level, when the smooth side-match method is used in the decoder of GTSVQ[Smooth] to search for the codeword with the smallest smooth side-match distortion as the decoding result. Therefore, the PSNR is enhanced in GTSVQ[Smooth] when the bit rate is low. Table II lists the percentages to which that the codewords encoded for each coding level are the same as that of the fifth coding level in GTSVQ[Smooth]. Meanwhile, Fig. 3 compares the coding quality of the various methods, and reveals that GTSVQ[Smooth] has better coding quality than other coding methods. Three reasons for this phenomenon exist. First, the algorithm CLUSTERING searches clusters more effectively than the algorithm LBG. GTSVQ[LBG] uses the LGB algorithm to design the general-tree coder as the same

TABLE II
PERCENTAGES TO WHICH THAT THE SEARCHED CODEWORDS IN FOUR CODING LEVELS ARE THE SAME AS THAT OF THE FIFTH CODING LEVEL IN THE GTSVQ[SMOOTH]

Images	Percentages (%)				
	(1)	(2)	(3)	(4)	(5)
Lena	81.2	89.5	95.3	98.8	100
F-16	78.6	86.2	94.3	97.2	100
Boat	82.2	90.8	96.2	99.1	100

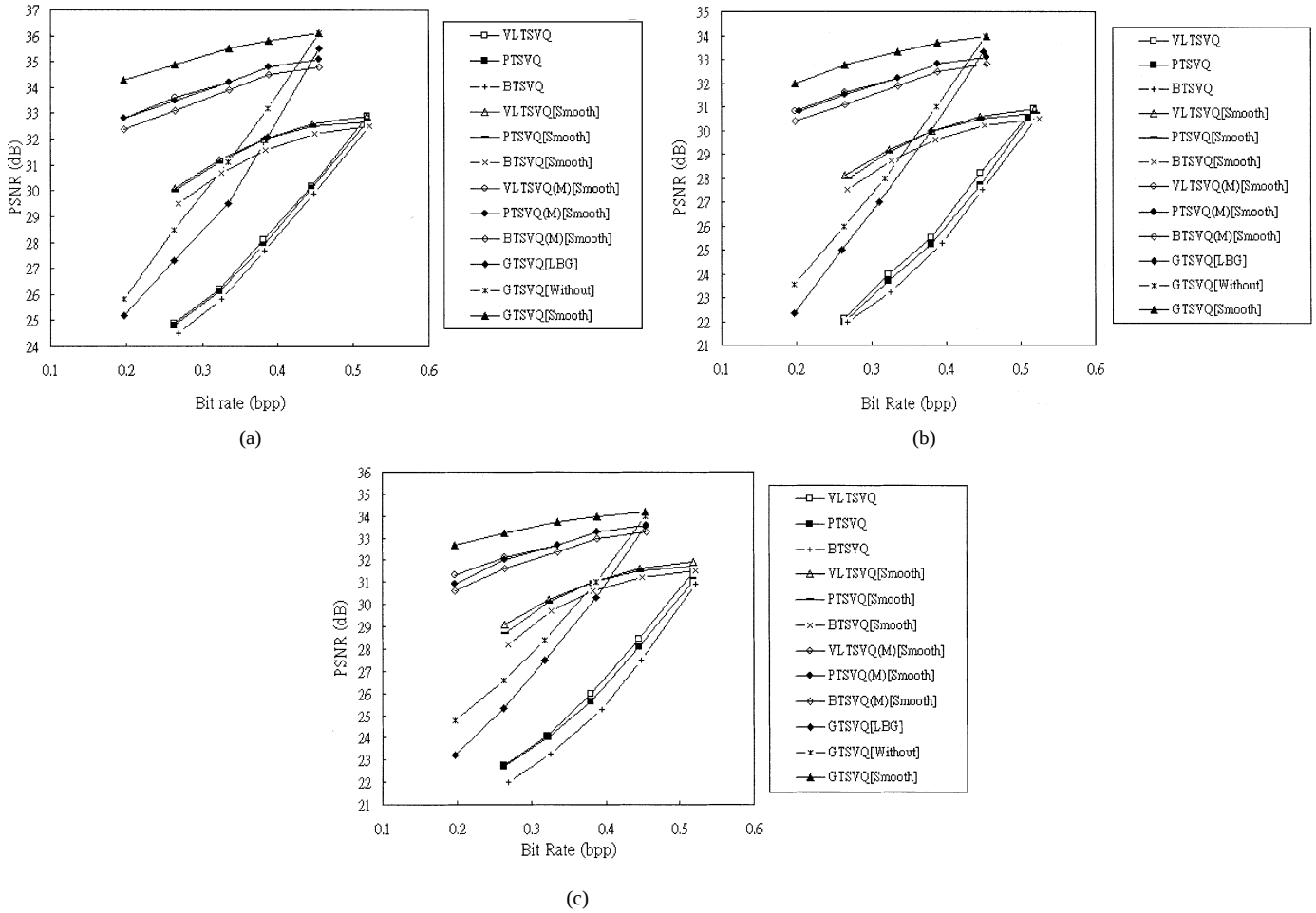

Fig. 3. Comparison of coding performance by the various coding methods. (a) Coding quality of the "Lena" image. (b) Coding quality of the "F-16" image. (c) Coding quality of the "Boat" image.

structure of GTSVQ[Without]. GTSVQ[Without] achieves higher PSNR in its coding quality than the GTSVQ[LBG]. Second, the decoder of GTSVQ[Smooth] uses the smooth side-match method in the second step to select a codeword with the smallest side-match distortion as the decoding result, thus achieving a better solution than only using first step in the decoder of GTSVQ[Smooth], as GTSVQ[Without] does. That is, when the bit rate is low, Fig. 4(a) and 4(b) display the coding results by using GTSVQ[Without] and GTSVQ[Smooth], respectively, when the first coding level is finished. Fig. 4 reveals that coding quality is enhanced when the smooth side-match method is used in GTSVQ[Smooth]. Third, GTSVQ generally achieves a higher search rate than conventional TSVQs based on a binary tree structure. Table III lists the search rates of the various coding methods, with search rate indicating the percentage to which that the searched codeword is the same as that of the full search. In the genetic algorithm, if w is small, the genetic algorithm tends to produce more clusters and the general tree widens, meaning that search time lengthens but PSNR tends to increase. Meanwhile, if w is large, the algorithm tends to produce fewer clusters and the general-tree will be narrower, meaning that searching time is reduced but so too is PSNR. The general-tree vector quantizer is, therefore, something between two extremes, the binary tree vector quantizer and the

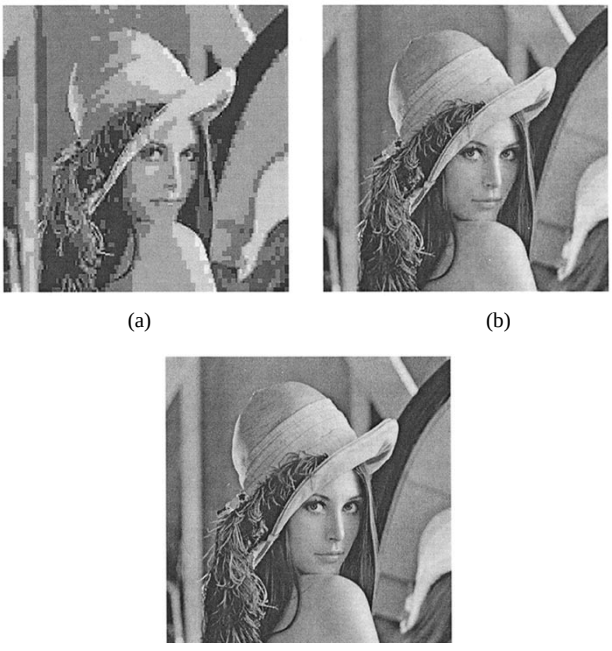

Fig. 4. "Lena" image encoded by using the methods GTSVQ(Without) and GTSVQ(Smooth). (a) Image encoded by the method GTSVQ(Without) (25.8 dB, 0.197 bpp). (b) Image encoded by the method GTSVQ(Smooth) (34.3 dB, 0.198 bpp). (c) Original "Lena" image.

TABLE III
SEARCH RATE OF THE COMPARISON OF OUR METHOD AND THE OTHER METHODS

Methods	Search rates(%)				
	(1)	(2)	(3)	(4)	(5)
VLTSVQ	98.5	96.4	94.1	91.9	89.2
PTSVQ	98.4	96.3	94.0	91.7	89.0
BTSVQ	98.2	96.0	93.8	91.4	88.5
VLTSVQ(M)	99.6	98.8	98.1	97.2	96.4
PTSVQ(M)	99.6	98.8	98.1	97.1	96.3
BTSVQ(M)	99.4	98.6	97.7	96.7	95.8
GTSVQ(Without)	99.8	99.1	98.5	97.8	97.2

full search vector quantizer. Consequently, the search rate of the proposed method exceeds that of conventional TSVQs.

VII. CONCLUSIONS

TSVQ has the advantage of more efficient codebook searches than traditional full-search vector quantizers. However, it is improper to always divide a set of training samples into a fixed number of clusters in TSVQ. This generally causes an increase in either the bit rate, or the average distortion, or both. A general-tree-structured vector quantizer is proposed herein. The CLUSTERING algorithm is used to divide a set of training samples into several natural clusters in accordance to the characteristics of the training data set. Following the construction of the general-tree coder, the Huffman coding is used to optimize the bit rate. Progressive coding can also be achieved by using the method designed herein. Moreover, the smooth side-match method is presented in this paper. Combining the Huffman tree decoder and the smooth side-match method to select the codewords in the decoder yields good coding quality at a lower bit rate. As evidenced by the experimental results, the proposed method achieves a higher PSNR and lower average bit rate than other methods when applied to the same data set.

REFERENCES

[1] Y. Linde, A. Buzo, and R. M. Gray, "An algorithm for vector quantizer design," *IEEE Trans. Commun.*, vol. 28, pp. 84–95, 1980.

[2] L. Y. Tseng and S. B. Yang, "Genetic algorithms for clustering, feature selection and classification," in *Proc. IEEE Int. Conf. Neural Networks*, Houston, TX, June 1997.

[3] ——, "A genetic clustering algorithm for data with nonspherical-shape clusters," *Pattern Recognit.*, vol. 33, pp. 1251–1259, 2000.

[4] A. Buzo, A. H. Gray Jr., R. M. Gray, and J. Markel, "Speech coding based upon vector quantization," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 28, pp. 562–574, May 1985.

[5] J. Makhoul, S. Roucos, and H. Gish, "Vector quantization in speech coding," *Proc. IEEE*, vol. 73, pp. 1551–1588, Nov. 1985.

[6] P. A. Chou, T. Lookabaugh, and R. M. Gray, "Optimal pruning with applications to tree-structured source coding and modeling," *IEEE Trans. Inform. Theory*, vol. 35, pp. 299–315, Feb. 1989.

[7] E. A. Riskin and R. M. Gray, "A greedy tree growing algorithm for the design of variable rate vector quantizers," *IEEE Trans. Signal Processing*, vol. 39, pp. 2500–2507, Nov. 1991.

[8] M. Balakrishnan, W. A. Pearlman, and L. Lu, "Variable-rate tree-structured vector quantizers," *IEEE Trans. Inform. Theory*, vol. 41, pp. 917–930, Apr. 1995.

[9] T. D. Chiueh, T. T. Tang, and L. G. Chen, "Vector quantization using tree-structured self-organizing feature maps," *IEEE J. Select. Areas Commun.*, vol. 12, Sept. 1994.

[10] U. Bayazit and W. A. Pearlman, "Variable-length constrained-storage tree-structured vector quantization," *IEEE Trans. Image Processing*, vol. 8, pp. 321–331, Mar. 1999.

[11] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees. The Wadsworth Statistics/Probability Series*. Belmont, CA: Wadsworth, 1984.

[12] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1990.

[13] T. Kim, "Side match and overlap match vector quantizers for images," *IEEE Trans. Image Processing*, vol. 1, pp. 170–185, Feb. 1992.

[14] R. F. Chang and W. T. Chen, "Image coding using variable-rate side-match finite-state vector quantization," *IEEE Trans. Image Processing*, vol. 2, no. 1, pp. 104–108, 1993.

[15] T. S. Chen and C. C. Chang, "A new image coding algorithm using variable-rate side-match finite-state vector quantization," *IEEE Trans. Image Processing*, vol. 6, pp. 1185–1187, Aug. 1997.

[16] H. C. Wei, P. C. Tsai, and J. S. Wang, "Three-sided side match finite-state vector quantization," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 10, pp. 51–58, Jan. 2000.

[17] D. S. Kim and S. U. Lee, "Image vector quantizer based on a classification in the DCT domain," *IEEE Trans. Commun.*, vol. 39, no. 4, pp. 549–556, 1991.

[18] M. H. Lee and G. Crebbin, "Classified vector quantization with variable block-size DCT models," in *Proc. IEEE Visual Image Signal Processing*, vol. 141, 1994, pp. 39–48.

[19] J. Vaisey and A. Gersho, "Image compression with variable block size segmentation," *IEEE Trans. Signal Processing*, vol. 40, pp. 2040–2060, 1992.

[20] P.-Y. Yin and L.-H. Chen, "A new noniterative approach for clustering," *Pattern Recognit. Lett.*, vol. 15, no. 2, pp. 125–133, 1994.

[21] L. Y. Tseng and S. B. Yang, "A genetic approach to the automatic clustering problem," *Pattern Recognit.*, vol. 34, pp. 415–424, 2001.