International Journal of Innovative
Computing, Information and Control
Volume 7, Number 10, October 2011

ICIC International ©2011 ISSN 1349-4198
pp. 5961–5976

LOSSLESS INDEX CODING FOR IMAGE VECTOR QUANTIZATION USING HUFFMAN CODES

Hung-Min Sun and Bying-He Ku

Department of Computer Science
National Tsing Hua University
No. 101, Sec. 2, Kuang-Fu Road, Hsinchu 30013, Taiwan
hmsun@cs.nthu.edu.tw

Received June 2010; revised December 2010

Abstract. *Image vector quantization (VQ) has many current and future envisioned applications, such as digital image and signal compression, watermarking, data hiding, speaker identification. The idea of lossless index coding is to enhance the performance of image VQ by further encoding the index map of the VQ without introducing extra distortion. A novel lossless index coding scheme, called Index Associated List Coding (IALC), is proposed in this paper. When we encode the given current index, the first part of our scheme searches for an equivalent index among the neighboring indices. If such an index is found, the current index can be encoded with bits that represent the position of the equivalent index. Furthermore, we propose a coding structure, called Index Associated List (IAL), to encode the current index efficiently. The performance of IALC is evaluated by comparing it with three famous lossless index coding schemes: SOC, LCIC and LVIC. IALC outperforms the other three schemes in terms of average bit rate. Compared with conventional image VQ, IALC reduces the bit rate from 0.50 b/pixel to 0.26 b/pixel with a codebook size of 256.*
Keywords: Image compression, Vector quantization, Huffman codes, Image coding, Index compression

1. **Introduction.** Image vector quantization (VQ) is a well-studied technique that has been applied to digital image and signal compression [1, 2, 3, 4, 5, 6, 7, 8]. Image VQ has research value due to its envisioned applications, such as digital watermarking [9, 10, 11], data hiding [12, 13, 14], speaker identification [15, 16], image retrieval [17] and clustering [18, 19]. For example, the main idea behind VQ-based digital watermarking schemes is to carry watermark bits via codeword indices. VQ-based digital watermarking schemes have applications in copyright protection and image authentication. Moreover, many VQ-based data hiding schemes have been proposed recently to embed secret data into the VQ indices of the cover image according to some characteristics of hidden data.

Image VQ can be classified into memoryless image VQ and memory image VQ. In memoryless image VQ, the image to be encoded is divided into a set of blocks represented by vectors. Each vector is quantized (compressed) by searching the most similar codeword in the codebook independently. Instead of storing the image vector, we store the index of the winning codeword (i.e., the most similar codeword). However, performing a full-search is very time-consuming; thus, many fast search algorithms have been proposed in recent years [20, 21, 22, 23]. Most of the proposed fast searching techniques search for the winning codeword within small candidate areas instead of performing a full-search. Baek et al. [21] presented a projection-based fast encoding algorithm. In their approach, the potential power was computed by using projection axis. The scope of searching the winning codeword could be narrowed according to the potential power. Pan et al. [22]

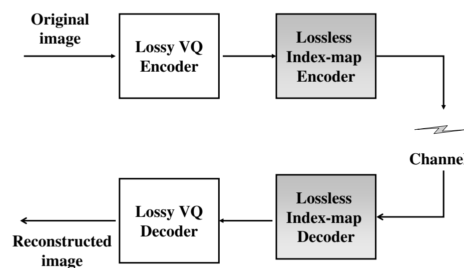


Figure 1. Lossless index coding process

proposed a 3-step hierarchical fast search algorithm, where sums and partial sums are used as features to measure the approximate difference between an input image block and a codeword among a sum-sorted code list. Their simulations demonstrated that searching the winning codewords with the features differences greatly reduces the computation time. In addition, they presented another estimation-based algorithm [23]. The authors introduced an additional new estimation for Euclidean distance and then optimized the computing with a full-search-equivalent algorithm [20].

With the image VQ decoder, the original image is recomputed, with small distortion, by using the received indices. The quality of the reconstructed image highly depends on the codebook size. Lower distortion can be achieved if a larger codebook is used; however, this will also require a higher bit rate. To reduce the distortion, Pan et al. [24] employ post-processing to encode poorly matched image blocks further after conventional image VQ.

Unlike memoryless image VQ which encodes each image vector independently, memory image VQ exploits the correlation between neighboring blocks in an image to improve the coding performance of conventional image VQ. There are highly correlated among the neighboring blocks when the block size is small, e.g., 4×4 block. Finite-State VQ (FSVQ) and Address VQ (AVQ) are memory image VQ methods, [25, 26, 27]. An FSVQ encoder consists of a finite set of sub-codebooks. The FSVQ encoder encodes the input vector by searching the winning codeword from the sub-codebooks [25, 27]. However, FSVQ introduces extra coding distortion. AVQ improves coding performance of conventional image VQ without introducing extra coding distortion by utilizing an address codebook [26]. The address codebook consists of a set of address codevectors where each codevector represents a combination of indices (addresses).

Novel lossless index-map coding schemes were studied and have attracted comprehensive attention recently [28, 29, 30, 31, 32]. Figure 1 shows the flow diagram of lossless index coding scheme. (Here, the lossless index coding means the original index-map of image VQ can be recovered exactly from its compressed version without introducing any extra coding distortion.) The idea of lossless index coding methods is to further encode the index map of conventional image VQ by exploiting neighboring index correlation in the index map. The advantage of these approaches is that computing inter-index correlation requires only scalar computations.

Hsieh et al. [28] first proposed a lossless index coding scheme called Search-Order-Coding (SOC). In SOC, each index in the index map was encoded by searching the same index from indices within a predefined range centered at the current index. If any previous index matches the current index, the current index is then replaced by the number of searching order. They assumed that the number of bits required to represent

the search order was smaller than the number of bits required to represent the original index. Based on their simulation results (shown in Table 3), the average bit rates of SOC were 0.29 b/pixel, 0.34 b/pixel and 0.41 b/pixel for codebooks of the size 128, 256 and 512, respectively. Hu et al. [32] presented a Low Complexity Index-Compressed (LCIC) approach to study the similarity between adjacent indices and the property of codebook ordering. In LCIC, the offset among the current index and its left adjacent index in the ordered codebook was computed. Then, the current index was encoded as the offset if this offset was smaller than a predefined threshold. Chen et al. [30] proposed Lossless Vector-quantized Index Coding (LVIC) for improving the coding efficiency further. LVIC constructed several adaptive coding trees initially and each index in the index map was encoded by a selected coding tree with two rules, called horizontal and vertical relation bit. Furthermore, the authors use memory bank to record some of previously encoded indices with respect to the current index.

Several papers mentioned above are the lossless index coding, and however, all utilize some short code to represent the encoding index which is found equivalent one among previous encoded neighboring indices. In fact, our observations by getting statistics several benchmark images suggest that there are probably half the indices which would not be matched with their previous encoded neighboring indices. To overcome this deficiency, we compress these indices by a special data structure (IALs) which is constructed with Huffman coding.

In this paper, we propose a high compression and lossless index coding scheme called Index Associated List Coding (IALC). To model the correlation among neighboring indices, we construct several Huffman trees, called Index Associated Lists (IALs). According to our observations in many images, the current processing index is highly correlated with its adjacent index. For example, the current index is equal to its left adjacent index value with high probability. To demonstrate IALC, we model the left-right correlation as several index associated lists. (Note that IALC can be applied to various inter-index correlations. Encoding the current index from its upper adjacent index is also feasible on IALC with a little modification.) Initially, we model left-right correlation information by statistically collecting the occurrence frequency of adjacent index pairs, i.e., (left index, the current processing index), with many representative images. Therefore, if an index has higher occurrence frequency along with its left index, it could be represented by fewer bits according to the concept of Huffman coding. Based on this observation, we construct an index associated list (IAL) for each index value with the collected data. In the encoding step, the current processing index is encoded by employing the corresponding IAL with respect to its left adjacent index. The details of the proposed index associated list coding (IALC) are described in Section 2. The experimental results show that IALC significantly improves compression performance of image VQ without extra coding distortion. Compared with the conventional image VQ, the proposed algorithm reduces the bit rate from 0.44 b/pixel to 0.21 b/pixel, 0.50 b/pixel to 0.26 b/pixel and 0.56 b/pixel to 0.32 b/pixel with codebook size 128, 256 and 512, respectively.

The rest of this paper is organized as follows: in Section 2, we present IALC algorithm in detail; experimental results are given in Section 3; finally, we conclude this paper in Section 4.

2. Index Associated List Coding Scheme (IALC).

To reduce the overhead of transmitting digital images in communication channel, we propose an index coding scheme to further compress the index map generated from conventional image VQ schemes. It has been shown in [28, 29, 30, 31, 32] that the current processing index is highly related to its neighboring indices. In this paper, the relationship between the current processing index

TABLE 1. The proportion of each condition in I_C's adjacent blocks with codebook size 128

Condition	Avg	
$I_1 = I_C = I_2$	21.20%	$f_1 = 38.12\%$
$I_1 = I_C \neq I_2$	16.92%	
$I_2 = I_C \neq I_1$	14.53%	$f_2 = 14.53\%$
$I_1 \neq I_C, I_C \neq I_2$	47.34%	$f_3 = 47.34\%$

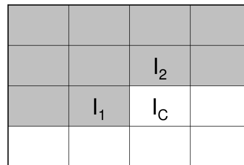


FIGURE 2. The position-relation of I_C and its adjacent indices I_1, I_2. Shadowed blocks represent previous encoded indices and white blocks denote the indices to be encoded.

and its left neighboring index is referred to as the left-right correlation. We will introduce a data structure, called index associated list (IAL), to represent the left-right correlation information.

2.1. **Index associated list coding (IALC).** Assume that the index associated lists (IALs) are given. In this subsection, we will illustrate how to encode the index map of image VQ using the given IALs. We delay the presentation of the IALs until Section 2.2. The proposed IALC starts to encode the index map from the top left index in a raster scan order. Let I_C represent the current processing index in the index map. Let I_1 and I_2 be the left adjacent and the upper adjacent indices of the current processing index I_C, respectively as shown in Figure 2. The index relation among I_1, I_2, I_C that require consideration can be summarizes with 3 cases: (1) $I_1 = I_C$, (2) $I_1 \neq I_C$, $I_C = I_2$ and (3) $I_1 \neq I_C, I_C \neq I_2$. IALC encodes the three cases separately with different coding strategies.

2.1.1. *The design of prefix code tree.* To distinguish (at the decoder) the three cases mentioned above, a "prefix code" is needed to be appended to the forefront of each encoding result. For this reason, we designed a "prefix code tree" using Huffman coding scheme. In Table 1, f_1, f_2 and f_3 represent the occurrence frequencies of cases (1)-(3) respectively. Table 1 is obtained by collecting samples from five benchmark images with codebook size 128. The reader should notice that although we only demonstrate the result of codebook size 128, this observation is still held up when the codebook size is 256 and 512 respectively.

Table 1 suggests that the highest occurrence frequency is case (3), and the second highest is case (2). According to this information, we design a prefix code tree in Figure 3. Although, we have stated above that neighboring indices are very similar, Table 1 suggests the current index processing has only near $(1 - 47.34\%)$ probability to be equal to its left or upper adjacent indices. This is why conventional lossless index coding

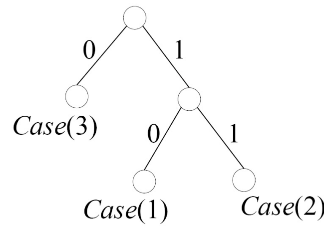


FIGURE 3. Prefix code tree

approaches, such as SOC [28], LCIC [32] and LVIC [30], can further improve their bit rates. The basic idea of IALC is first to search an index that is equivalent to the current processing index from neighboring indices. If such an index is found, the current index can be encoded with fewer bits; otherwise, they can be encoded efficiently with Huffman coding.

2.1.2. *The proposed coding scheme.* IALC encodes and decodes the index map one by one in raster scan order. In the decoding stage, the decoder has known I_1 and I_2 before decoding I_C. In other words, if the decoder knows that I_C must be equal to either I_1 or I_2 according the prefix code, then the two conditions, ($I_1 = I_C$ and $I_1 = I_2$) and ($I_1 \neq I_C$ and $I_C = I_2$), are sufficient to distinguish case (1) from case (2). That is, the decoder can reconstruct I_C according to the prefix code and the two conditions. Therefore, with the prefix code tree, the IALC encodes the current processing index as "10" if $I_1 = I_C$. If $I_1 \neq I_C, I_C = I_2$, the current processing index is encoded as "11". Otherwise, "$0||H(I_1, I_C)$" is encoded. Here $H(I_1, I_C)$ denotes the code returned by the Index Associated List (IAL), and "$0||H(I_1, I_C)$" means the string concatenation of "0" and $H(I_1, I_C)$. We will postpone the illustration of $H(I_1, I_C)$ until Section 2.3.

Consider the condition $I_1 = I_C, I_1 = I_2$, which is originally contained in case (1) and should be encoded as "10" as described above. However, the current processing index that satisfies this condition can be encoded with a shorter code without any ambiguity during decoding time. This is because that I_1 and I_2 are known information before decoding I_C. The current processing index can be encoded as "1" independently if $I_1 = I_C$ and $I_1 = I_2$ are satisfied.

Thus, we subdivided case (1) into two cases, $I_1 = I_C, I_1 = I_2$ and $I_1 = I_C, I_1 \neq I_2$, denoted by case (1.a) and case (1.b) respectively. When the current processing index satisfies the condition $I_1 = I_C$, the decoder only needs to perform an additional check to see if $I_1 = I_2$. If the condition $I_1 = I_2$ is satisfied, the current processing index satisfies case (1.a), we can encode the index as "1", otherwise, we encode the index as "10". The detailed coding strategies used in IALC is shown in Table 2. In summary, the current processing index is encoded as "1" if $I_1 = I_C, I_1 = I_2$, as "10" if $I_1 = I_C$, as "11" if $I_1 \neq I_C, I_C = I_2$, and as "$0||H(I_1, I_C)$" if otherwise. We assume that both sender (encoder) and receiver (decoder) have the same codebook and IALs. The proposed IALC algorithm is shown in Algorithm 1.

Table 2. The conditions and the coding strategies for $Case(i)$

$Case(i)$	Condition	Coding Strategy
$Case(1.a)$	$I_1 = I_C$, $I_1 = I_2$	1
$Case(1.b)$	$I_1 = I_C$, $I_1 \neq I_2$	10
$Case(2)$	$I_1 \neq I_C$, $I_C = I_2$	11
$Case(3)$	$I_1 \neq I_C$, $I_C \neq I_2$	$0\|\|H(I_1, I_C)$

Algorithm 1 Index Associated List Coding

Require: The index map of image VQ.
Ensure: The compressed index map S.

```
1: S ← the binary representation of the top left index
2: repeat
3:    I_C ← the next index in raster scan order
4:    if I_C = I_1 = I_2 then {Case (1.a)}
5:       S ← S||1
6:    else if I_C = I_1 then {Case (1.b)}
7:       S ← S||10
8:    else if I_C = I_2 then {Case (2)}
9:       S ← S||11
10:   else {Case (3)}
11:      S ← S||0||H(I_1, I_C)
12:   end if
13: until I_C = φ
14: return S
```

2.2. **Index associated list (IAL).** Based on the observations in several experiments, the current processing index would possibly equal to certain index values according to its left adjacent index. This relationship, called left-right relationship, is helpful to compress the current processing index efficiently. To further compressing the left-right relationship in images, an optimum and instantaneous encoding method is needed. By assigning code words with variable lengths according to variable probabilities of these left-right relationship, Huffman coding results in minimum-redundancy codes.

The Index Associated List (IAL) is a Huffman tree that models the correlation between the current processing index and its left adjacent index. An IAL is constructed with respect to a codevector of the codebook. Thus, there are totally N IALs for a codebook of size N. Let IAL(k) denote the IAL with respect to the codeword k. The IAL(k) is constructed using all codevectors in the codebook excluding codevector k itself. In other words, each IAL has $N-1$ leaf nodes.

To design the Huffman tree for IAL(i), the occurrence frequency of each of the $N-1$ codewords (excluding for the codeword i) needs to be determined in advance. Let $f(i,j)$ denotes relative appearing frequency, $1 \leq i$, $j \leq N$. The current processing index has index value j and its left adjacent index has value i. In other words, for a fixed i, $f(i,j)$ indicates how many times that $I_C = j$ appears when $I_1 = i$.

We measure the relative appearing frequency $f(i,j)$ by employing some benchmark images. Therefore, an IAL(i) for $1 \leq i \leq N$ can be constructed with Huffman coding [33] with input $f(i,j)$, for $1 \leq j \leq N$ and $j \neq i$, as illustrated in Figure 4. We denote

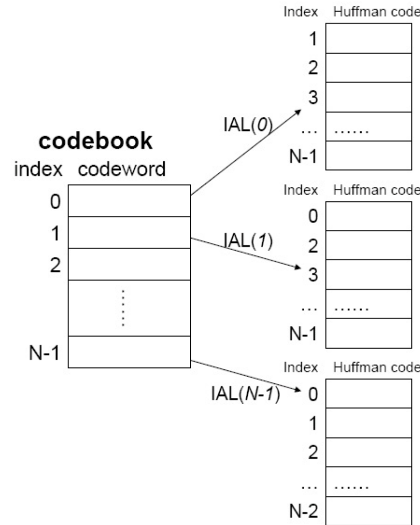


FIGURE 4. The illustration of index associated lists

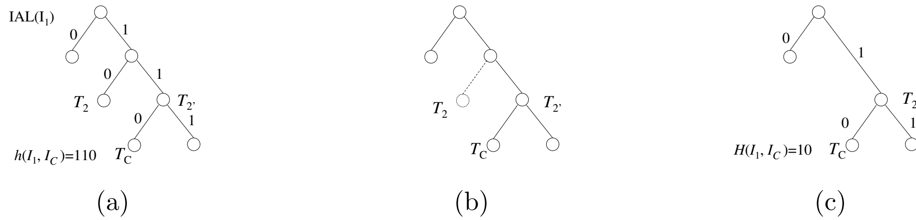


FIGURE 5. The sketch of eliminating one bit from $h(I_1, I_C)$ in the case (3)

$h(I_1, I_C)$ the bit streams returned by searching for I_C in $IAL(I_1)$. In Section 2.3, we will show how to enhance the constructed IALs.

2.3. $\boldsymbol{H(I_1, I_C)}$. Assume that T_k is the node with respect to the index I_k in the IAL(I_1), and $Code(T_k)$ is Huffman code of T_k. According to IALC algorithm (see Algorithm 1), in the case of $I_1 \neq I_C, I_C \neq I_2$, I_C is encoded as "$0||H(I_1, I_C)$". However, in this case we never have to search I_2 in IAL(I_1) because we know $I_C \neq I_2$ in advance. This information is useful because if allows us to remove one bit from $h(I_1, I_C)$, when T_C is one of the descendants of the sibling node of T_2. Let $T_{2'}$ be the sibling node of T_2 and $a_i \in \{0, 1\}$. Without loss of generality, we assume that

$$Code(T_2) = a_1 a_2 \cdots a_u. \tag{1}$$

Since T_2 and $T_{2'}$ are sibling nodes, the last one position u of $Code(T_{2'})$ differs from that of $Code(T_2)$. In other words, we can express $Code(T_{2'})$ as follows:

$$Code(T_{2'}) = a_1 a_2 \cdots \overline{a_u}. \tag{2}$$

Let $D(T_{2'})$ be set of descendants of $T_{2'}$. For any $T_p \in D(T_{2'})$, $Code(T_p)$ can be expressed as:

$$Code(T_p) = a_1 a_2 \cdots a_{u-1} \overline{a_u} a_{u+1} \cdots a_m. \tag{3}$$

According to the information "$I_C \neq I_2$", we can remove the bit $\overline{a_u}$ from $Code(T_p)$ and obtain the shorter code $Code_{new}(T_p)$ as follows:

$$Code_{new}(T_p) = a_1 a_2 \cdots a_{u-1} a_{u+1} \cdots a_m. \tag{4}$$

(Note that we merely remove one bit from $Code(T_p)$, rather than modifying the Huffman tree IAL(I_1).)

Formally, $H(I_1, I_C)$ is defined as:

$$H(I_1, I_C) = \begin{cases} Code_{new}(T_C) & \text{if } T_C \in D(T_{2'}), \\ h(I_1, I_C) & \text{otherwise.} \end{cases} \tag{5}$$

Given a bit stream, we will show how to correctly decode the bit stream in Section 2.5.

Take Figure 5 as an illustrative example. Figure 5(a) is an illustrative IAL tree in which $Code(T_2) = 10$ and $Code(T_{2'}) = 11$. $Code(T_C) = h(I_1, I_C) = 110$. Since T_C is a descendent node of $T_{2'}$ and $I_C \neq I_2$, we can "hide" the second bit "1" to represent T_C as "10", as shown in Figure 5(b) and Figure 5(c). In other words, when the encoder discovers that both the conditions are true, the decoder decodes the current index I_C as "10" directly rather than "110". The reader should note that Figure 5(b) and Figure 5(c) are used for illustrative purpose. The original IAL(I_1) is not modified during the elimination procedure.

2.4. **The processing of boundary indices.** IALC algorithm encodes an index (I_C) by referencing its left adjacent index I_1 and its upper adjacent index I_2. However, every index in the upper boundary of an image lacks upper adjacent index, and every index in the left boundary of an image lacks left adjacent index. Thus, additional processing is required for these indices in the upper boundary and the left boundary.

1. Upper boundary: We imagine that I_2 exists and assume that "$I_1 = I_2$" for all indices in the upper boundary of an index map. In other words, if $I_1 = I_C$, then the current processing index is encoded by case (1.a) in the IALC strategy. Otherwise, if $I_1 \neq I_C$, then the coding strategy in case (3) is used.
2. Left boundary: Similar to the modified strategy for the upper boundary of an index map, we deal with the left boundary by assuming that I_1 exists and "$I_1 = I_2$" for all indices in the left boundary.

2.5. **The decoding scheme for IALC.** Given a bit stream $S = s_1 s_2 s_3 \cdots s_n$, $s_i \in \{0, 1\}$. Similar to the encoder, the decoder decodes the index map index by index in raster scan order. The decoding algorithm of IALC is illustrated in Algorithm 2. The purpose of Algorithm 2 is to convert the binary bit stream S into a decimal index map I, in which parameters c and i represent the current position in S and the current index to be decoded in the index map I respectively. Assume that $I = \{I(1), I(2), \cdots, I(wh)\}$ is the set of indices arranged in raster scan order, where w and h represent the width and height of the index map. If the decoding index is not in the boundary, then $I(i-1)$ and $I(i-w)$ indicate the its left and upper neighboring indices. The first index (the top-left corner index in the map) can be decoded directly with the first $\log_2 N$ bits in S, where N is the size of the codebook. Then the first index will be a prior information for decoding the second index and so forth.

Algorithm 2 Index Associated List decoding

Require: The bit stream S.
Ensure: The index map I of image VQ.

```
 1: c ← log N, i ← 1, I(i) ← the decimal representation of (s_1 ··· s_c)
 2: repeat
 3:   c ← c + 1, i ← i + 1
 4:   I_1 ← I(i − 1), I_2 ← I(i − w)
 5:   if B_i = 1 then {in the boundary}
 6:     if s_c = "1" then
 7:       if i ≤ w then {upper boundary}
 8:         I(i) ← I_1
 9:       else {left boundary}
10:         I(i) ← I_2
11:       end if
12:     else
13:       (I(i), j) ← R_Index(c + 1, I_1, I_2, S), c ← c + j
14:     end if
15:   else {B_i = 0, not in the boundary}
16:     if s_c = "1", I_1 = I_2 then {Case (1.a)}
17:       I(i) ← I_1
18:     else if s_c s_{c+1} = "10" then {Case (1.b)}
19:       I(i) ← I_1, c ← c + 1
20:     else if s_c s_{c+1} = "11" then {Case (2)}
21:       I(i) ← I_2, c ← c + 1
22:     else { Case (3)}
23:       (I(i), j) ← R_Index(c + 1, I_1, I_2, S),c ← c + j
24:     end if
25:   end if
26: until c = n
27: return I
```

In Algorithm 2, we use a flag "B_i" to denote whether the current index $I(i)$ is in the boundary, i.e., $B_i = 1$. For $1 \leq i \leq wh$,

$$B_i = \begin{cases} 1 & \text{if } i \leq w \text{ or } i (\bmod w) = 1, \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

For those indices in the upper and left boundary, the decoder first checks whether the first one bit is "1". When this case occurs, the current index is decoded by assigning $I_C = I_1$ or $I_C = I_2$. Otherwise, the current index is decoded by case (3) (lines 6-14 in Algorithm 2).

If I_C is not in the boundary ($B_i = 0$), the decoder checks if $I_1 = I_2$ and the top one bit in S to determine which one prefix code is utilized. When the top one bit in S is "0", the current index $I(i)$ satisfies the condition in case (3), and $I(i)$ can be decoded by traversing the tree IAL(I_1) (lines 22-24); otherwise, $I(i)$ will be decoded by reversing the encoding steps according to Table 2 (lines 16-21).

Algorithm 3 R_Index(t, I_1, I_2, S)

```
1: if I_1 = I_2 then
2:    (I_C, j) ← Traverse(t, S, I_1)
3: else
4:    a_1 a_2 ··· a_u ← h(I_1, I_2)
5:    if s_t s_{t+1} ··· s_{t+u-2} = a_1 a_2 ··· a_{u-1} then
6:       S ← s_1 s_2 ··· s_{t+u-2} || ā_u || s_{t+u} ··· s_n
7:       (I_C, j) ← Traverse(t, S, I_1)
8:    else
9:       (I_C, j) ← Traverse(t, S, I_1)
10:   end if
11: end if
12: return (I_C, j)
```

Algorithm 3 is used to decode the current index $I(i)$ when the condition in case (3) occurs. As we described in Section 2.3, there is one bit to be removed from $h(I_1, I_C)$ if $I_1 \neq I_2$ and T_C is one of the descendants of the sibling node of T_2 in the case of (3). According to Equations (1) and (2), T_C is one of the descendants of the sibling node of T_2 if and only if T_C and T_2 have the same bit stream "$a_1 a_2 \cdots a_{u-1}$". Therefore, the decoder can determine that one bit was removed from $h(I_1, I_C)$, i.e., the decoder checks if the bit stream "$s_{c+1} s_{c+2} \cdots s_{c+u-1}$" matches with "$a_1 a_2 \cdots a_{u-1}$", which is obtained by searching $h(I_1, I_C)$. When the decoder discovers that one bit was removed from $h(I_1, I_C)$, S can be recovered by adding back the last one bit of $h(I_1, I_2)$ (lines 5-7 in Algorithm 3). In Algorithm 3, Traverse(t, S, I_1) is denoted as the procedure to get the current index by traversing a Huffman tree with a series of bits. It takes three inputs: (1) t: the position of the starting bit, (2) S: the bit stream S and (3) I_1: the Huffman tree to be traversed. Then, Traverse(t, S, I_1) generates two results (I_C, j): (1) I_C: the current index and (2) j: the number of bits used to reach a leaf node.

2.6. **Complexity analysis.** The time and space complexity of the proposed IALC encoding and decoding algorithms are described, respectively. Since the codebooks and the IALs are off-line construction, we did not include the cost of its construction on our analysis.

2.6.1. *The time complexity of the encoding process.* Given an index map $I(i)$, $1 \leq i \leq w \times h$ and the codebook with size N, the time complexity of Algorithm 1 is considered as follows.

Obviously, lines 1-9 in Algorithm 1 have time complexity $O(1)$. In line 11, as we introduced in Section 2.3, $H(I_1, I_C)$ has two operations: (1) obtains $h(I_1, I_C)$ and (2) checks if there is one bit that could be removed. Obtaining $h(I_1, I_C)$ involves finding the Huffman code of I_C in IAL(I_1). This operation can be achieved within time complexity $O(1)$ if IAL(I_1) is implemented as a Huffman table sorted by index. The second operation, checking if there is one bit that could be removed, involves matching two Huffman codes, $h(I_1, I_2)$ and $h(I_1, I_C)$, respectively. In the worst case, the length of a Huffman code is $N - 1$. Thus, the time complexity of computing $H(I_1, I_C)$ is

$$O(1) + O(N) = O(N). \tag{7}$$

Since the repeat loop runs at most $w \times h - 1$ times, the overall time complexity of Algorithm 1 is

$$O(1) + (w \times h - 1)O(N) = O(whN). \tag{8}$$

2.6.2. *The time complexity of the decoding process.* The decoding process of IALC contains Algorithm 2 and Algorithm 3. To facilitate the analysis, we first analyze the time complexity of Algorithm 3. In line 2 of Algorithm 3, the Traverse() function is run to search an index in an IAL tree with a series of input binary bits. Since the depth of an IAL tree is at most $N-1$, performing Traverse() function requires time complexity $O(N)$. Similarly, line 7 and line 9 require time complexity $O(N)$. Since the length of a Huffman code is at most $N-1$, comparing two Huffman codes in lines 5-6 of Algorithm 3 takes time complexity $O(N)$. Thus the overall time complexity of Algorithm 3 is $O(N)$.

In Algorithm 2, all lines obviously takes time complexity $O(1)$ excluding line 13 and line 26. Lines 13 and 26 execute Algorithm 3 to compute the value of the current index I_C by searching the IAL(I_1) with a series of binary bits. Since each move in the IAL(I_1) tree costs one bit and the length of the input binary bit stream is at most n, lines 13 and 26 will be repeated $O(n/N)$ times. Thus, the overall complexity of Algorithm 2 is $O(n/N \cdot N) = O(n)$.

2.6.3. *The space complexity of the encoding and decoding process.* All the encoder and the decoder and the IALs need to store the same codebook. In general, the size of a codeword in the codebook is so small, e.g., 4×4 or 8×8, that we can regard it as constant. The space complexity of constructing the codebook is $O(N)$. As we described in Section 2.2, the number of IAL tree is N. Furthermore, there are $N-1$ codewords in each IAL(i), $1 \leq i \leq N$. Suppose each IAL(i) is implemented with a Huffman table. In the worst case, every entry in the Huffman table needs $N-1$ bits to store its Huffman code. Thus, the space complexity of IALs is

$$O(N(N-1)(N-1)) = O(N^3). \tag{9}$$

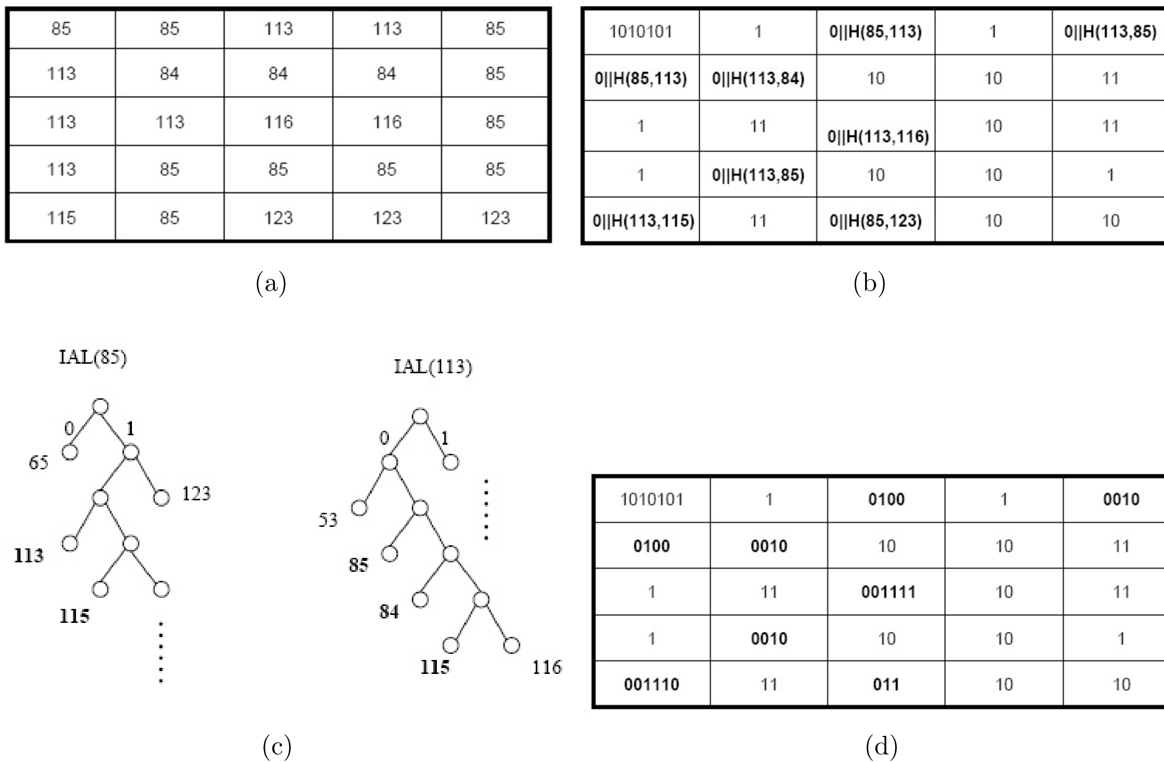
The overall space complexity for storing both the codebook and the IALs is

$$O(N) + O(N^3) = O(N^3). \tag{10}$$

The storage required here is the same with the storage required in encoding process. The decoder needs codebook and IALs to do the decoding process. Therefore, the space complexity of Algorithm 2 is $O(N^3)$.

2.7. **An example for IALC.** Here, we use an index map of size 5×5, shown in Figure 6(a) to illustrate the proposed IALC approach. In this example, the codebook size is 128, and thus an index in the codebook is represented by $\log_2 128$ bits. The top left index 85 is first encoded using the binary representation of 85, i.e., "1010101", as illustrated in Figure 6(b) and Figure 6(d).

Since the encoder encodes the index map in raster scan order, the next index to be encoded is 85. Because the index 85 is in the upper boundary, we assume its upper adjacent index is equal to its left adjacent index 85, i.e., $I_1 = I_2 = 85$. Thus we encode this index as "1" according to the coding strategy in case (1.a) in IALC algorithm. The third index to be encoded is 113. Similarly, we assume $I_1 = I_2 = 85$, and encode the third index 113 as "0$||H(85, 113)$" by using the coding strategy in case (3). By looking up the index 113 in IAL(85) shown Figure 6(c), we obtain $H(85, 113) =$ "100". Therefore, the index 113 is encoded as "0100". Let us skip the illustration of encoding the fourth, fifth and sixth indices, and focus on the seventh index 84. In this case, $I_2 = 85$ and $I_1 = 113$. This index satisfies the condition in case (3) and should be encoded as "0$||H(113, 84)$". According to the Figure 6(c), we will obtain $h(113, 84) =$ "0110" by searching for the leaf node 84 in IAL(113). Because leaf node 84 is a descendant of the sibling node of leaf node 85, we can remove the second bit "1" from "0110". Consequently, index 84 is encoded as



85	85	113	113	85
113	84	84	84	85
113	113	116	116	85
113	85	85	85	85
115	85	123	123	123

(a)

<table>
<tr><td>1010101</td><td>1</td><td>0||H(85,113)</td><td>1</td><td>0||H(113,85)</td></tr>
<tr><td>0||H(85,113)</td><td>0||H(113,84)</td><td>10</td><td>10</td><td>11</td></tr>
<tr><td>1</td><td>11</td><td>0||H(113,116)</td><td>10</td><td>11</td></tr>
<tr><td>1</td><td>0||H(113,85)</td><td>10</td><td>10</td><td>1</td></tr>
<tr><td>0||H(113,115)</td><td>11</td><td>0||H(85,123)</td><td>10</td><td>10</td></tr>
</table>

(b)

(c)

1010101	1	0100	1	0010
0100	0010	10	10	11
1	11	001111	10	11
1	0010	10	10	1
001110	11	011	10	10

(d)

FIGURE 6. An example of IALC: (a) the index map to be encoded, (b) the illustrative encoding result of Algorithm 1, (c) IAL(85) and IAL(113), (d) the final output bit stream

“0010”. The final output bit stream is $S =$ “1 0 1 0 1 0 1 1 0 1 0 0 1 0 0 1 0 0 1 0 0 0 0 1 0 1 0 1 0 1 1 1 1 1 0 0 1 1 1 1 1 0 1 1 1 0 0 1 0 1 0 1 0 1 0 0 1 1 1 0 1 1 0 1 1 1 0 1 0”.

Given a bit stream S shown above, we demonstrate the decoding procedure as follows. Initially, the counter $c = 1$. Let s_i denote the i-th bit in S, e.g., $s_1 = 1$ and $s_2 = 0$. In the beginning, $\log_2 128 = 7$ bits, i.e., “1010101”, are extracted from the bit stream to recover the top left index. In this example, the top left one index 85 is first recomputed and c is increased by 7, i.e., $c = 8$.

The next bit to be checked is $s_8 =$ “1”. As we described in Section 2.4, we assume that $I_1 = I_2$ for those indices in the upper boundary. Thus, we can obtain the second index 85 with the coding strategy in case (1.a). The counter c is then increased by 1.

Since $s_9 =$ “0” and $I_1 = 85$, according to case (3), the decoder uses the bits that follow s_9 to traverse the tree IAL (85) shown in Figure 6(c) and recover the third index 113. The counter c in this case is increased by 4.

Again, we skip the decoding process of the fourth, fifth and sixth indices, and concentrate on decoding the seventh index. Now, let us set the counter c to be 22. In this case, the decoder knows $I_1 = 113$ and $I_2 = 85$ before decoding the seventh index. Since $s_{22} =$ “0” and $I_1 \neq I_2$, the current processing index is decoded with the coding strategy in case (3). Before using the bits that follow s_{22} to traverse the tree IAL(113), the decoder needs to search IAL(113) for the index $I_2 = 85$ and determine its Huffman code “010”. The decoder knows there is one bit removed by the encoder from comparing the first two bits in “010” to s_{23} and s_{24}. Thus the decoder will add one bit “1” which is the complement of the third bit in “010” to traverse IAL(113). In other words, the decoder first uses $s_{23} =$ “0” and $s_{24} =$ “1” followed by this additional bit “1”, and finally, $s_{25} =$ “0” to

Table 3. Experimental results of the image quantities using different vector quantization methods (bpp)

size	RAR	ZIP	ARJ	SOC	LCIC	LVIC	IALC
128	0.33	0.34	0.35	0.29	0.29	0.25	0.21
256	0.40	0.42	0.43	0.34	0.37	0.32	0.26
512	0.50	0.51	0.52	0.41	0.46	0.40	0.32

recover the seventh index 84. We can reconstruct the original index map without loss by continuing this process until the final bit in the bit stream is reached.

3. Experimental Results and Discussion.

3.1. Experimental results.

In this section, we evaluate the performance of our IALC technique by comparing it with three other famous lossless index coding schemes: SOC [28], LCIC [32] and LVIC [30]. Our experiments employ three codebooks with sizes 128, 256 and 512. The three codebooks are generated by using well-known LBG algorithm [4]. The images in the USC-SIPI image database [34] are used as test images. Both the constructions of the codebooks and IAL are off-line. The bit rate of IALC scheme is evaluated as follows.

$$BR = \Big[(Case(1.b) + Case(2)) \cdot 2 + (Case(1.a) + Case(3)) \cdot 1 + \sum b_j\Big] / (w \cdot h \cdot s) \tag{11}$$

- $Case(i)$: The number of index values that satisfies the condition in case (i) shown in the second column of Table 2.
- b_j: The number of bits required to encode index j if j satisfies the condition in case (3) (Note that b_j is set to be zero if index j satisfies the condition in case (1.a), (1.b) or (2).)
- h, w: The height and width of the image, respectively.
- s: The codevector size.

The experimental results are shown in Table 3. Clearly, IALC outperforms the other three approaches no matter what codebook size is used. Among these three previously proposed methods, LVIC exhibits the best compression ratio, and our technique improves LVIC by 16% average bit rate with codebook size 128. Furthermore, IALC shows slightly better compression ratio as the codebook size increases, e.g., IALC improves LVIC by 18% and 19% average bit rate with codebook size 256 and 512, respectively. Comparing with the conventional image VQ, IALC reduces the bit rate from 0.44 b/pixel to 0.21 b/pixel, 0.50 b/pixel to 0.26 b/pixel and 0.56 b/pixel to 0.32 b/pixel when codebooks size is 128, 256 and 512, respectively.

Moreover, we compare IALC with some well-known commercial lossless compressors such as ZIP, RAR and ARJ. ZIP and ARJ are the systems based on the adaptive Huffman coding and LZ77. RAR is the system uses a combination of statistical, entropy, and dictionary based compressors. From the results shown in Table 3, it illustrates that when the codebook size is bigger, the bit rate of IALC is smaller than these commercial compressors in average.

TABLE 4. The proportion of encoding indices with original indices

size	SOC	LCOC	LVIC	IALC
128	32%	36%	31%	43%
256	41%	47%	40%	52%
512	52%	58%	51%	63%

TABLE 5. The average length in $H(I_1, I_C)$ used in IALC (bits)

size	bits
128	4.8
256	5.5
512	6.2

3.2. **Discussion.** The idea behind the three approaches, SOC, LCIC and LVIC, is that they all attempt to search for an index which is equivalent to the current processing index. If they successfully obtain one, the current processing index can be encoded with fewer bits than the original index. However, if they fail to find such an equivalent index, the penalty is that they have to represent the current processing index as the original index plus prefix codes. However, according to our observations in many experiments, up to thirty to fifty percent of indices cannot be encoded with fewer bits in the three approaches, as shown in Table 4. The core of our technique is to encode these indices that fail to discover an equivalent index with Huffman coding method. Table 5 shows the average length needed to represent an index with Huffman coding. The result show that we can obtain better results with $H(I_1, I_C)$ rather than with original indices. (4.8, 5.5 and 6.2 bits are better than $\log 128, \log 256$ and $\log 512$ bits, respectively.)

In addition, the prefix code tree of IALC is also constructed with Huffman coding. IALC uses only one bit as the prefix code in case (3) and case (1.a), where the proportion of both cases (3) and (1.a) is about $47.34\% + 21.20\% = 68.54\%$ (see Table 1). In case (1.b) and case (2), IALC uses two bits as the prefix code, and the proportion of both cases is about $16.92\% + 14.53\% = 31.45\%$. On average, the length of a prefix code in IALC method is $1 \cdot 68.54\% + 2 \cdot 31.45\% = 1.31$ bits. As compared with uniform bit allocation which costs two bits for each case in IALC, our design of prefix code tree reduces 0.69 bits per index.

4. **Conclusion.** Research on image VQ is necessary due to its envisioned applications, such as digital watermarking, data hiding, speaker identification. Exploring inter-index correlation had been shown by previous research studies to be an efficient solution for improving image VQ performance because it involves only scalar computation. This paper presents a novel data structure, called index associated list (IAL), to model the inter-index correlation information. To demonstrate the proposed IALC, this paper employs the left-right correlation as a case study and evaluate the performance of the proposed IALC method. The experimental results shows that IALC has a better compression ratio than previously presented lossless index coding approaches, such as SOC, LCIC and LVIC. Significantly, our IAL structure is compatible with the other three methods; thus, the IAL can also be utilized by the three approaches to encode those indices that cannot discover equivalent index in close proximity to the current processing index. As compared with

the conventional image VQ, IALC reduces the bit rate from 0.44 b/pixel to 0.21 b/pixel, 0.50 b/pixel to 0.26 b/pixel and 0.56 b/pixel to 0.32 b/pixel when codebooks size is of 128, 256 and 512, respectively.

Acknowledgment. This work was supported in part by the National Science Council, Taiwan, under Contract NSC 97-2221-E-007-055-MY3 and NSC 99-2218-E-007-012. The authors also gratefully acknowledge the helpful comments and suggestions of the reviewers, which have improved the presentation.

REFERENCES

[1] A. Gersho and R. M. Gray, *Vector Quantization and Signal Compression*, Kluwer Academic Publishers, 1992.
[2] R. M. Gray, Vector quantization, *IEEE Trans. Information Theory*, vol.28, pp.157-166, 1982.
[3] N. M. Nasrabadi and R. A. King, Image coding using vector quantization: A review, *IEEE Trans. Commun.*, vol.36, pp.957-971, 1998.
[4] Y. Linde, A. H. Buzo and R. M. Gray, An algorithm for vector quantizer design, *IEEE Trans. Commun.*, vol.28, pp.84-95, 1980.
[5] R. Y. Li, J. Kim and N. A. Shamakhi, Image compression using transformed vector quantization, *Image and Vision Computing*, vol.20, pp.37-45, 2002.
[6] K. Kang, A fast and dynamic region-of-interest coding method based on the patterns in jpeg2000 images, *International Journal of Innovative Computing, Information and Control*, vol.5, no.5, pp.1161-1170, 2009.
[7] Q. Wang, D. Zhao and W. Gao, Context-based adaptive variable length coding for video dct blocks, Part I – Theoretical approach, *ICIC Express Letters*, vol.3, no.2, pp.141-146, 2009.
[8] Q. Wang, D. Zhao and W. Gao, Context-based adaptive variable length coding for video dct blocks, Part II – Practical scheme, *ICIC Express Letters*, vol.3, no.2, pp.163-170, 2009.
[9] F. H. Wang, L. C. Jain and J. S. Pan, VQ-based watermarking scheme with genetic codebook partition, *Journal of Network and Computer Applications*, vol.30, pp.4-23, 2007.
[10] C. Lin, J. S. Pan and B. Y. Liao, Robust VQ-based digital image watermarking for mobile wireless channel, *IEEE International Conference on Systems, Man and Cybernetics*, vol.3, pp.2380-2384, 2006.
[11] Z. M. Lu, D. G. Xu and S. H. Sun, Multipurpose image watermarking algorithm based on multistage vector quantization, *IEEE Transactions on Image Processing*, vol.14, pp.822-831, 2005.
[12] C. C. Chang, Y. P. Hsieh and C. Y. Lin, Lossless data embedding with high embedding capacity based on declustering for VQ-compressed codes, *IEEE Transactions on Information Forensics and Security*, vol.2, pp.341-349, 2007.
[13] C. C. Chang and W. C. Wu, Hiding secret data adaptively in vector quantisation index tables, *IEE Proc. of Vision, Image and Signal Processing*, vol.153, pp.589-597, 2006.
[14] W. C. Du and W. J. Hsu, Adaptive data hiding based on VQ compressed images, *IEE Proc. of Vision, Image and Signal Processing*, vol.150, pp.233-238, 2003.
[15] G. Zhou and W. B. Mikhael, Speaker identification based on adaptive discriminative vector quantisation, *IEE Proc. of Vision, Image and Signal Processing*, vol.153, pp.754-760, 2006.
[16] T. Kinnunen, E. Karpov and P. Franti, Real-time speaker identification and verification, *IEEE Transactions on Audio, Speech and Language Processing*, vol.14, pp.277-288, 2006.
[17] A. M. E. Moghadam, J. Shanbehzadeh, F. Mahmoudi and H. S. Zadeh, Image retrieval based on index compressed vector quantization, *Pattern Recognition*, vol.36, pp.2635-2647, 2003.
[18] E. Lughofer, Extensions of vector quantization for incremental clustering, *Pattern Recognition*, vol.41, pp.995-1011, 2008.
[19] Q. She, H. Su, L. Dong and J. Chu, Support vector machine with adaptive parameters in image coding, *International Journal of Innovative Computing, Information and Control*, vol.4, no.2, pp.359-368, 2008.
[20] J. Mielikainen, A novel full-search vector quantization algorithm based on the law of cosines, *IEEE Signal Processing Letters*, pp.175-176, 2002.
[21] S. J. Baek, B. K. Jeon and K. M. Sung, A fast encoding algorithm for vector quantization, *IEEE Signal Processing Letters*, vol.4, pp.325-327, 1997.

[22] Z. Pan, K. Kotani and T. Ohmi, A hierarchical fast encoding algorithm for vector quantization with PSNR equivalent to full search, *IEEE Proc. of Int. Symp. Circuits and Systems*, vol.1, pp.797-800, 2002.

[23] Z. Pan, K. Kotani and T. Ohmi, An improved full-search-equivalent vector quantization method using the law of cosines, *IEEE Signal Processing Letters*, vol.11, pp.247-250, 2004.

[24] Z. Pan, K. Kotani and T. Ohmi, A post-processing method for vector quantization to achieve higher PSNR and nearly constant bit rate, *IEEE Proc. of Int. Symp. Circuits and Systems*, vol.2, pp.436-439, 2003.

[25] J. Foster, R. Gray and M. Dunham, Finite-state vector quantization for waveform coding, *IEEE Trans. Information Theory*, vol.31, pp.348-359, 1985.

[26] N. M. Nasrahadi and Y. Feng, Image compression using address-vector quantization, *IEEE Trans. Commun.*, vol.38, pp.2166-2173, 1990.

[27] P. Yahampath and M. Pawlak, On finite-state vector quantization for noisy channels, *IEEE Signal Processing Letters*, vol.4, pp.325-327, 1997.

[28] C. H. Hsieh and J. C. Tsai, Lossless compression of VQ index with search-order coding, *IEEE Trans. Image Processing*, vol.5, pp.1579-1582, 1996.

[29] G. Shen and M. L. Liou, An efficient codebook post-processing technique and a window-based fast-search algorithm for image vector quantization, *IEEE Trans. Circuits and System for Video Technology*, vol.10, no.6, pp.990-997, 2000.

[30] P. Y. Chen and C. T. Yu, Lossless vector-quantised index coding design and implementation, *IEE Proc. of Circuits Devices Syst.*, vol.152, no.2, pp.109-117, 2005.

[31] P. Y. Chen and R. D. Chen, An index coding algorithm for image vector quantization, *IEEE Trans. Consumer Electronics*, vol.49, no.4, pp.1513-1520, 2003.

[32] Y. C. Hu and C. C. Chang, Low complexity index-compressed vector quantization for image compression, *IEEE Trans. Consumer Electronics*, vol.45, no.1, pp.198-201, 1999.

[33] S. Roman, *Introduction to Coding and Information Theory*, Springer-Verlag, 1997.

[34] *The Usc-sipi Image Database*, http://sipi.usc.edu/database/database.cgi?volume=misc.