Optimal Huffman Coding of DCT Blocks

Gopal Lakhani, *Senior Member, IEEE*

Abstract—It is a well-observed characteristic that, when a discrete cosine transform block is traversed in the zigzag order, ac coefficients generally decrease in size and the runs of zero coefficients increase in length. This paper presents a minor modification to the Huffman coding of the JPEG baseline compression algorithm to exploit this characteristic. During the run-length coding, instead of pairing a nonzero ac coefficient with the run-length of the preceding zero coefficients, our encoder pairs it with the run-length of subsequent zeros. This small change makes it possible for our codec to code a pair using a separate Huffman code table optimized for the position of the nonzero coefficient denoted by the pair. These position-dependent code tables can be encoded efficiently without incurring a sizable overhead. Experimental results show that our encoder produces a further reduction in the ac coefficient Huffman code size by about 10%–15%.

Index Terms—Discrete cosine transform (DCT), Huffman coding, JPEG image compression.

I. INTRODUCTION

DISCRETE cosine transform (DCT)-based compression algorithms have been proposed for image and video compression and conferencing systems such as JPEG, MPEG, and H.263. The JPEG baseline compression system [1] is perhaps the most widely used DCT-based system. In this system, the input image is partitioned into blocks of 8×8 pixels first and then each block is transformed using the forward DCT. Next, DCT coefficients are normalized using a preset quantization table and, finally, the normalized coefficients are entropy encoded. JPEG provides two entropy coding methods—arithmetic and Huffman coding. This paper deals with the Huffman coding of ac coefficients. A complete description of the baseline algorithm is given in [2] and details of the Huffman coding are given in [3, Sect. F.1.2].

The biggest advantage of using the DCT is that it packs the image data of a block into an almost optimal number of decorrelated coefficients, resulting in significant compression. Nonzero DCT coefficients are generally located close to the top/left corner of the transformed block. Further, the nonzero ac coefficients along the zigzag order [4, Fig. 9.3] decrease in size and runs of zero coefficients increase in length. Thus, DCT blocks possess a different kind of statistical redundancy not present in image blocks. The purpose of this paper is to exploit this redundancy for further compression by modifying the Huffman coding of the JPEG baseline algorithm. Our goal is to use multiple code tables, possibly one for each ac coefficient position.

A. Adaptive Huffman Coding

There are several ways to develop adaptive coding to improve the compression efficiency of JPEG Huffman coding and still use a single, fixed code table for all ac coefficients. Clearly, the easiest choice is to first develop a custom code table for the given image and then use it to code the image; JPEG provides this as an option. Table I presents experimental results, which show that use of the custom code table reduces[1] the code size further, but only by 1.38%. If the size of the custom table is taken into account, because the encoder must include the code table with the image code, there may not be any reduction. Another choice is to follow dynamic Huffman coding, [5], proposed for general file compression. In this method, the encoder updates the code table after coding each symbol to model any changes in the distribution of source symbols. Table I also contains experimental results, which show that this method obtains a further reduction of about 1% only; it is hardly appealing knowing that dynamic Huffman coding is highly computational. The reason for dynamic Huffman coding not achieving any further compression in our experiments is that there is hardly any change in the global distribution of ac coefficients; all changes are local and confined within each DCT block.

Literature on adaptive Huffman coding of image and/or video data is limited. An interesting variation of Huffman coding is given in [6]; it assigns code words dynamically, i.e., given the code table, symbols are assigned different codes from the same table as their frequency change; however, the table remains the same. A study to resolve mismatches between statistical characteristics and variable-length code tables for H.263L is given in [7]. JPEG, MPEG, and H.263 also allow some sort of adaptive coding. For example, the JPEG sequential-mode encoder can use up to four ac code tables, but only one table can be used for a block. MPEG and H.263 provide separate tables for intramode and intermode coding.

B. Using Multiple Code Tables

Use of multiple code tables for DCT coding has been suggested in the past. For example, the author of [8] categorizes DCT blocks on the basis of their ac coefficients and then uses a different code table for each category, but still uses one table per block. Our objective is different; we want to code each block using multiple ac code tables, because an adaptive coding strategy alone cannot exploit the redundancy confined within each DCT block. We present the reason for using multiple tables using a figure (Fig. 1). This figure plots the probability of occurrences of pairs (1,1) at different positions of a DCT block; (1,1) denotes an ac coefficient of magnitude 1 that

[1]Throughout this paper, reduction is measured in terms of the ac Huffman code size.

TABLE I
COMPARING JPEG CODING WITH PROPOSED CODING

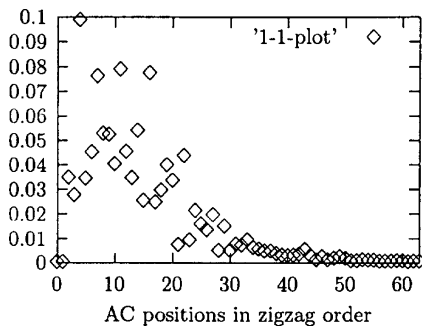| 1	2	3	4	5	6	7	8
Image name	JPEG Table K.5 code	JPEG Custom code	Dyna. Huff. code	Arith. Code	Our tables code	Table code cost	Our Saving
	bits	%	%	%	bits	bits	%
Sailboat	213173	0.29	0.14	0.81	184787	6278	10.97
Tiffany	150839	3.04	2.28	3.49	127740	7404	11.25
Pepper	162930	1.77	1.26	2.63	140232	6543	10.70
Lena	154984	1.15	0.69	2.12	134406	5977	10.25
Baboon	335662	0.73	0.53	0.66	281520	7101	14.38
Airplane	161470	0.63	0.31	1.43	141893	5901	9.27
Girl	249953	0.71	0.46	1.46	210357	5410	14.17
Gold	303798	0.90	0.67	1.93	259976	6251	12.78
Zelda	193689	3.46	2.79	1.43	162538	4684	14.30
Barbara	333950	1.15	1.00	1.33	286465	8201	12.14
Average	226045	1.38	1.01	1.93	192991	6376	12.02

Fig. 1. Distribution of the ac coefficient (1,1).

precedes exactly one zero coefficient. This pair is assigned a code of length four in the default JPEG ac code table. Assuming optimality of Huffman code tables, the probability of this pair should be $1/2^4$, i.e., 0.0625. Fig. 1 shows that the probability of (1,1) varies widely and, hence, use of a single code table for all ac positions is suboptimal.

Fig. 1 suggests that we should use a separate code table for each position to realize optimal coding. However, it creates a problem for the decoder, because a nonzero ac coefficient X is paired first with Z, a run of zero coefficients and then the pair (Z, X) is coded as a unit. The problem is that, without any knowledge of Z and hence, the position k of the coefficient X in the DCT block, the decoder would be unable to select the kth code table to decode the pair correctly. To make this point clear, let $0, 2, 0, 0, 1, \ldots$ denote ac coefficients of a block in the zigzag order or let $\{(1, 2), (2, 1), \ldots\}$ be the sequence of (Z, X) pairs. If the pair (1,2) is coded using code table 2, the position of coefficient 2 in the DCT sequence, the decoder would fail to select code table 2. There are two solutions to this problem:

1) Code a pair using the table that corresponds to the position of the first zero coefficient represented by the pair. If this solution is followed, the encoder will code the two nonzero coefficients occurring in positions 2 and 5 in the above sequence using the tables for positions 1 and 3, respectively, i.e., it will use nearby tables. While this solution may be satisfactory, it is clearly suboptimal. To illustrate this point, consider positions 9 and 10 in Fig. 1; the corresponding probabilities are approximately 0.05 and

0.04. The binary logarithm of the two numbers differ by about 1/3; it means that use of table 9 in place of table 10 may increase the code size by 1/3 bit. This suboptimality can be avoided using the following solution.

2) Change the order of Z and X, i.e., pair a nonzero ac coefficient with the run-length of subsequent zero coefficients. For example, for the above sequence, the encoder will code pairs $(0, 1), (2, 2), (1, *), \ldots$. Clearly, after decoding (0,1), the decoder would have no difficulty using the correct table for the pair (2,2).

This paper presents an implementation of the second solution. This solution requires developing a coding procedure, which should also handle the following two special situations not encountered in JPEG coding: 1) coding of initial run-length of zeros without pairing it with any nonzero coefficient, in case the block begins with zeros, and 2) coding of the last nonzero coefficient of a block without pairing it with trailing zeros. The next section describes this procedure. There are two advantages of the second solution: 1) a major advantage is that the codec can use a custom code table for each coefficient position and 2) a minor advantage, which is not immediately obvious, is that no end-of-block (EOB) marker is needed to separate the code of a DCT block from the next block.

To put the prospect of savings due to use of multiple ac code tables in perspective, we estimated sizes of different segments of the Huffman code by compressing several continuous-toned, photographic images of different characteristics at the default quality level. As per these experiments, on the average, 10.5% of the image code represents dc coefficients and 89.5% represent ac coefficients and EOB marker. Furthermore: 1) 34.8% of the ac code represents the sign/offset component and 2) 65.2% represents the Huffman table code. Since there is no correlation between the sign/offset of any two coefficients, there is no scope of reduction in portion 1), and any further reduction in 2) can be obtained only by revising the ac code tables or by improving the coding procedure; both are objectives of this paper. This is also the reason that we report only the ac Huffman table code bits and not the total image code bits in this paper.

This paper is organized as follows. Section II presents our DCT coding algorithm; it gives a detailed description of our code tables and how they are different from the JPEG code table.

Section III presents our method of coding position-dependent code tables, which is very different from the JPEG method. To reduce the table coding cost, it is possible not to code all tables; Section III also explores this possibility. Section IV presents the results of several experiments to compare the proposed coding method with JPEG Huffman coding.

II. DCT COEFFICIENT CODING ALGORITHM

A. Code Table Organization

We follow the Huffman coding algorithm of ac coefficients [3, Annex F.1.2.2]. This algorithm can be divided in two phases: 1) run-length coding and 2) code bit generation. We present here only the differences in the run-length coding phase. We represent a nonzero ac coefficient A followed by Z zeros by a pair (Z, X) directly, where X is the ac level of A.

For 8-b images, ac coefficients are confined to $[-1023, 1023]$ and, hence, $0 \leq X \leq 10$. If $Z > 15$, we split a pair into multiple pairs. The net result is that our ac code tables as well as the JPEG ac code table [3, Table K.5] can be expressed as two-dimensional (2-D) matrices with Z representing the rows and X representing the columns, $0 \leq Z \leq 15$ and $0 \leq X \leq 10$. However, there are 162 entries, $\{(Z, X) : 0 \leq Z \leq 15, 1 \leq X \leq 10; (15, 0), (0, 0)\}$ in Table K.5, whereas our code tables contain 176 entries, $\{(Z, X) : 0 \leq Z \leq 15, 0 \leq X \leq 10\}$. If X is nonzero, all tables are used the same way. All differences exist in the use of column 0 entries. These are as follows:

1) $(0,0)$ is a frequently used entry in Table K.5, since JPEG uses it as the EOB marker at the end of each block; we do not need any EOB marker. In our code table for position 1, $(0,0)$ is an infrequently used entry, and it is not an entry in other tables.
2) JPEG uses $(15,0)$ to indicate a run of 16 zeros; we use the same to denote a run of 15 zeros.
3) There are no other entries of the form $(Z,0)$ in Table K.5, where $0 < Z < 15$; we use them to code either the run-length of beginning zeros, or the last nonzero ac coefficient of a block.

The examples given at the end of the next subsection show use of these entries; they are also sufficient to develop our encoder.

B. Run-Length Coding

Let $\{a_1, a_2, \ldots, a_{63}\}$ denote the ac coefficient sequence of a block in the zigzag order. For simplicity, let a_i denote the ac level of the ith coefficient; therefore, $0 \leq a_i \leq 10$. Let $\{c_1, c_2, \ldots, c_N\}$ denote the positions of nonzero ac coefficients in the sequence. Our run-length encoder considers various cases and generates one or more pairs per nonzero ac coefficient as follows.

1) If N is 0, i.e., the block contains no nonzero ac coefficient, it generates $\{(0,0), (0,0)\}$.
2) If $N = 1$ and $c_1 = 1$, i.e., the only nonzero ac coefficient is at position 1, it generates $\{(0,0), (a_1, 0)\}$.
3) In general, it generates

$$\{(c_1 - 1, 0), (c_2 - c_1 - 1, a_{c_1}), \ldots, (c_{i+1} - c_i - 1, a_{c_i}), \ldots,$$
$$(c_N - c_{N-1} - 1, a_{c_{N-1}}), (a_{c_N}, 0)\}.$$

It generates $(c_1 - 1, 0)$ only if $c_1 > 1$ to denote the initial zero run-length. If $c_1 > 16$, this pair is split into two or more pairs such as $(c_1 - 16, 0)$ (15, 0). Likewise, if $c_{i+1} - c_i > 16$, $(c_{i+1} - c_i - 1, a_{c_i})$ is split into multiple pairs such as $(c_{i+1} - c_i - 16, a_{c_i})(15, 0)$. The last pair $(a_{c_N}, 0)$ denotes the last nonzero coefficient and all trailing zeros of the block.

Example 1: Let $\{2, 0, 2, 0, 0, 1, 1, 0, \ldots, 0\}$ be the ac sequence of a block. For this block, the JPEG encoder will generate pairs $\{(0,2) (1,2) (2,1) (0,1) (0,0)\}$, and our encoder will generate $\{(1,2) (2,2) (0,1) (1,0)\}$. Thus, we generate one less pair if the block begins with a nonzero ac coefficient, which is typical.

Example 2: Let $\{0, 0, 2, 0, \ldots, 0\}$ be the ac sequence. JPEG will generate $\{(2, 2) (0, 0)\}$, and we will generate $\{(2,0) (2, 0)\}$. We generate the same number of pairs, if the first ac coefficient is zero.

Example 3: Let $\{0^{15}, 2, 0^{15}, 2, 0^{16}, 1, 0^{13}, 1\}$ be the ac sequence, where 0^n denotes a run of n zeros. JPEG will output $\{(15,2) (15,2) (15,0) (0,1) (13,1)\}$, and we will output $\{(15,0) (15,2) (1,2) (15,0) (13,1) (1,0)\}$. Here, we generate an extra pair, but this happens only when a block contains a run of exactly 15 zeros, a rare situation.

C. Run-Length Decoder

This section presents pseudocode of our decoder program and the JPEG decoder given as a flow chart in [3, Fig. F. 13]. The intent is to show that: 1) our Huffman encoder also runs in one pass like the JPEG encoder and 2) our decoder incurs no extra run-time complexity. To follow the pseudocode, let CT denote the index of the next code table and let *Decode()* decode the next pair from the input code stream. If CT is 0, it decodes the dc coefficient.

```
CT = 0.
while (not end of input)
{
Decode(CT).
CT = 1.  DCT[1..63] = 0.  (Z, X) = Decode(CT).
if (Z, X) is (0,0) {(Z1, X1)  =  Decode(CT).
DCT[1] = Z1.  CT = 0.}
else
while (CT > 0)
{
if X = 0
if (CT = 1 or Z = 15)  CT = CT + Z.
else {DCT[CT] = Z.  CT = 0.}
else {DCT[CT] = X.  CT = (CT + Z + 1) (mod
64).}
if (CT > 0)  (Z, X) = Decode(CT).
}
}
```

The JPEG decoder program is as follows.

```
CT = 1.  DCT[1..63] = 0.  (Z, X) = Decode().
while (CT > 0)
```

TABLE II
CODE WORD SIZE TABLES FOR TWO SUCCESSIVE AC POSITIONS

	0	1	2	3	4	.	.	10	0	1	2	3	4	.	.	10
0	16	2	3	4	16	.	.	16	15	3	3	5	16	.	.	16
1	5	4	5	16	16	.	.	16	5	3	5	16	16	.	.	16
2	7	16	16	16	16	.	.	16	6	16	16	16	16	.	.	16
3	16	16	16	16	16	.	.	16	16	16	16	16	16	.	.	16
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
15	16	16	16	16	16	.	.	16	16	16	16	16	16	.	.	16

```
{
if  X = 0
if  (Z = 15)  CT = CT + 16.
else  CT = 0.
else  {DCT[CT + Z] = X.  CT = (CT + Z + 1)  (mod
64).}
if  (CT > 0)  (Z, X) = Decode().
}
```

III. CODING OF HUFFMAN AC CODE TABLES

A. Differential Coding

Our encoder should attach 63 code tables, one for each ac position, as a part of the input to the decoder. The problem is that, if we code each table individually using the JPEG table coding method [3, Sect. K.3.3.2], it will cause a big overhead. Therefore, we developed a new method which exploits characteristics of ac code tables for 8-b images and relationship that exists between the code tables for two successive ac positions.

We divide the Huffman code table generation procedure into two phases: 1) computation of the code word size and 2) assignment of bit sequences so that the code words form a prefix set. We assume that the encoder and decoder follow the same procedure for phase 2 and therefore, our encoder needs to provide only the output of the phase 1 to the decoder, i.e., the tables of code word sizes of pairs (Z, X).

For each ac position, given the frequency of pairs (Z, X) as input, our encoder runs the Huffman code tree algorithm and computes the set of code word size of all pairs; the set satisfies Kraft's inequality [4]. It stores this set as a matrix of size 16×11. Since the maximum code word size in Table K.5 is 16, we also impose this restriction and modify the code word size set so that all words of size greater than 16 are set to 16 and Kraft's inequality is still satisfied. In practice, this modification has virtually no effect on the optimality of the Huffman code tables. The code word size matrices posses certain characteristics, which are very useful for table coding. These are: 1) each entry is less than or equal to 16; 2) Since the frequency of (Z, X) is higher for smaller Z and X, smaller entries (i.e., more significant code words) are located in the top/left part of each matrix; 3) the frequency of (Z, X) changes very gradually with the ac position, which means that entry (Z, X) in the matrix for the $(i + 1)^{\text{th}}$ position can be specified by coding the difference with the same entry of the matrix for the ith position, where $i = 1, 2, \ldots, 63$.

To illustrate above, Table II shows sample matrices for some two successive ac positions. Table III is the difference matrix; most of its entries are zeros, which are denoted using *.

TABLE III
DIFFERENCE OF CODE WORDS SIZES

Z	0	1	2	3	4	.	.	10
0	-1	1	0	1	*	.	.	*
1	0	-1	*	*	*	.	.	*
2	-1	*	*	*	*	.	.	*
3	*	*	*	*	*	.	.	*
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.
15	*	*	*	*	*	.	.	*

To code a difference matrix, we code its columns in order with the 0th column first. For each column, we code first the number of entries to be encoded from the column and then code the entries. Since most nonzero entries are located in the top/left part of a difference matrix, not all columns need to be coded. The position of the last coded column is specified by coding a 0. For this reason, we encode at least one entry from each previous column, even if they contain no nonzero entry.

Thus, for Table III, our encoder will generate the following:

$$h(3)h(2)h(1)h(1)h(0)h(-1)h(0)h(-1)h(+1)h(-1)h(0)h(+1)$$

where $h()$ denotes the Huffman code of its argument. The first four items denote that the number of entries coded from the first four columns are 3, 2, 1, and 1, respectively. The next item $h(0)$ denotes that no entries are coded from the remaining columns. The remaining items denote column entries. The argument to $h()$ is actually an unsigned integer; a signed argument simply denotes that the code is followed by a sign bit, e.g., $h(+1)$ is $h(1)$ followed by bit 0. The encoder provides the code table $h()$ to the decoder by encoding the corresponding code word set. Since the maximum number of entries in a column is 16 and an entry of a difference matrix is an integer between -15 and 15, we can realize the code word set of $h()$ using an array of size 17. Therefore, 17 bytes is also added to the overhead cost of our tables.

B. Reduction of the Number of Code Tables

Since there is an overhead cost associated with each ac code table, a goal of our encoder should be to not use a separate table for each ac position. We formulate next a minimization problem to address this issue.

Let $P_i(Z, X)$ denote the count coded at position i for the whole input image, where $0 \le i \le 63$, $0 \le Z \le 15$, and $0 \le X \le 10$. Let $C_i(Z, X)$ denote the code word size of (Z, X) in the code table for position i. Let $Tcost(k, i)$ denote the number of bits needed to code the difference matrix for position i, where the difference is taken with the code size matrix for position k, $k < i$. Let $\{i_t\}$ be an increasing sequence of integers in the
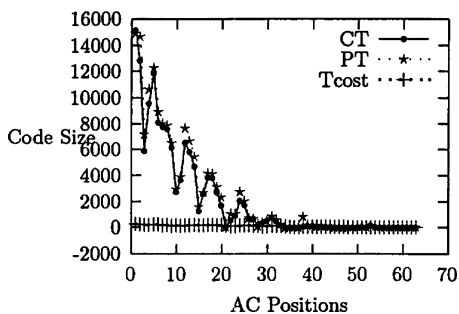
Fig. 2.   AC code size and table cost.

interval [1,63], which also denotes the positions of code tables used by the encoder. In this situation, given an ac coefficient at position l so that $i_t \leq l < i_{t+1}$, the encoder would use the code table for position i_t, $1 \leq t \leq n$. Clearly, $i_1 = 1$, $i_n \leq 63$, and $n < 63$. Let $i_0 = 0$ and $i_{n+1} = 64$. Given the sequence $\{i_t\}$, the total code size including the overhead cost of code tables can be expressed as follows:

$$\sum_{t=1}^{n} \sum_{l=i_t}^{i_{t+1}-1} \sum_{Z=0}^{15} \sum_{X=0}^{10} P_l(Z,X)*C_{i_t}(Z,X) + \sum_{t=0}^{n-1} Tcost(i_t, i_{t+1}).$$

Since the purpose of reducing the number of code tables is to reduce the overhead cost, perhaps at the expense of increased ac code size, it is important that we compare the overhead cost of the code tables with the ac code size generated at different positions. For this purpose, in Fig. 2, we show three plots:

1) ac code size generated at position i by using the code table for position i; it is $\sum_{Z=0}^{15} \sum_{X=0}^{10} P_i(Z,X) * C_i(Z,X)$ and it is labeled, CT (current table).

2) ac code size generated at position i but by using the code table for position $i - 1$; in this case, the code table for position i is not coded, and the code size is computed as $\sum_{Z=0}^{15} \sum_{X=0}^{10} P_i(Z,X) * C_{i-1}(Z,X) - Tcost(i-1, i)$; this is labeled PT (previous table).

3) the overhead of the table for position i coded by taking the difference with the table for position $(i - 1)$; it is $Tcost(i-1, i)$.

These plots are given for the Lena image.

Fig. 2 shows that Tcost is very small for all ac positions. It also shows that PT is above CT at most positions. Further, for high-frequency coefficient positions, PT and CT are near zero, i.e., any ac code reduction, which is obtained due to the use of PT in the place of CT for a position, is miniscule, and it is likely to happen for high-frequency positions only. Experimental results given in the next section show that the overhead cost of all code tables constitutes only about 3.3% of the ac code size. For this reason and the fact that if our encoder uses the same code table for two consecutive ac positions, this information can be provided to the decoder by using $h(0)$ alone, we decided not to develop any heuristics to solve the minimization problem stated above.

IV. EXPERIMENTAL RESULTS

To compare results of our encoder with the JPEG Huffman encoder, we present two sets of experimental results in this sec-

tion. The objective of the first set, given in columns 2–5 of Table I, is to show that Table K.5 is indeed very efficient. Hence, first we compare results of JPEG coding with some adaptive as well as nonadaptive Huffman coding methods that also use only one ac code table. Column 2 gives the ac table code size (in bits) obtained by using Table K.5. Column 3 presents reduction in terms of percentage of the code size obtained by using a custom ac code table for each image over column 2. Column 4 gives reductions obtained by dynamic Huffman coding program of [10]. Column 5 presents results of a nonadaptive, character-mode arithmetic encoder [11], which encodes each pair (Z, X) as a single source symbol.

The average of each column 3–5 is in the range of 1%–2%. The closeness of these results are sufficient to conclude that JPEG Table K.5 is quite efficient and that the column 2 numbers are near minimum, provided that images are coded using only one ac code table.

The second set of results, given in columns 6–8, is to measure the performance of the proposed coding method. Column 6 gives the ac table code size, and column 7 gives the overhead cost in bits of all ac code tables coded by our method. Column 8 gives the percentage reduction over column 2.

Before comparing the JPEG Huffman coding with our method (i.e., columns 2 or 3 with columns 6–7), we should point out that columns 2–3 do not include the overhead cost of the code table. The number of bits needed to code the code word size set of Table K.5 as per [3, Sect. K.3.3.2] is 1432. We expect this number to be close to 1432 for a custom-code table also. Therefore, first we compare column 6 with 2 and 3 directly; on average, column 6 is 14.62% smaller than column 2 and 13.0% smaller than column 3. To consider the overhead cost as well, we add 1432 to column 2 and then compare with columns 6 and 7 together. In this case, the average ac code reduction due to use of position-dependent code tables is 12.02%.

Comparing columns 7 with 6, we find that the overhead of all 63 code tables is only 3.30% of the total ac table code size. This means that, if we code larger images, say of size 2K × 2K, this ratio should drop to one fourth.

Table I presents results of high bit-rate image coding (the quality level Q of [9] was set to 75). To compute similar results for medium and low bit-rate coding, we compressed images by setting Q to 50 and 30, respectively. Since Table K.5 is known to perform poorly at lower bit rates, it was not used. Instead, a single custom code table and position-dependent tables were derived for each image. Results are not given for individual images; only the average of JPEG custom code (column 3), code using our tables (column 6), and the overhead cost (column 7) are reported here. For $Q = 50$, these are 167423, 144359, and 5200, respectively, and for $Q = 30$, these are 104184, 91574, and 3846. As before, we assume that it takes 1432 bits to code a single JPEG ac table. Thus, the reduction due to use of position-dependent code tables is 10.48% and 8.94% when Q is set to 50 and 30, respectively. Comparing the results for $Q = 75$ and $Q = 30$, we find that the overhead cost does not drop by the same proportion as the drop in the ac code size. The reason is that it takes more bits to code tables for low-frequency posi-

tions and these tables are still coded even if the coding bit rate is low.

Next, we compare savings of our method with the results given in [12], but first we give a brief summary of its method, which is an adaptation of Golomb–Rice coding for DCT coefficients. In [12], the context of a DCT block is computed based on the difference of its dc coefficient with the dc of the four neighboring blocks. Further, the dc range is quantized to a number of levels, resulting in a large number of block contexts. Within each context, the average of each of 63 ac coefficients is kept as the image is coded and this information is used to code DCT coefficients. Thus, this method also requires maintaining a large number of tables. The advantage of [12] is that no tables need to be coded as a part of the image code, which we do, but the overhead cost of our code tables is very small. On the other hand, Golomb–Rice coding is much more complex than Huffman coding. The work in [12] does not give reduction to ac coefficients separately. For images compressed at $Q = 70$, it reports a reduction of about 10%. The reduction reported in Table I is thus comparable with that in [12]

V. SUMMARY AND CONCLUSION

In this paper, we have focused on a statistical redundancy present in the ac sequence of DCT blocks. It is confined within DCT blocks and varies locally. As a result, adaptive coding methods such as dynamic Huffman coding or arithmetic coding, which consider global distribution of input symbols, do not obtain much further reduction. To exploit this redundancy, we propose that instead of pairing a nonzero ac coefficient with the run-length of preceding zeros, the run-length encoder should pair it with the run-length of subsequent zeros. The major advantage of this change is that the encoder can now use a custom code table optimized for each ac coefficient position. The proposed modification does require that the encoder also include position-dependent code tables with the image code. For this reason, we also proposed an efficient method for encoding the ac code tables. A minor advantage of our coding method is that no EOB marker is needed to represent the end of a block.

REFERENCES

[1] W. B. Pennebaker and J. L. Mitchell, *JPEG Still Image Data Compression*. New York: Van Nostrand Reinhard, 1993.

[2] G. K. Wallace, "The JPEG still picture compression standard," *Commun. ACM*, vol. 34, pp. 30–44, Apr. 1991.

[3] "Digital Compression and Coding of Continuous-Tone Still Images,", ISO DIS 10 981 part 1.

[4] J. D. Gibson, T. Berger, T. Lookabaugh, D. Lingburgh, and R. L. Baker, *Digital Compression for Multimedia*. San Francisco, CA.: Morgan Kaufmann, 1998.

[5] D. E. Knuth, "Dynamic Huffman coding," *J. Algorithms*, vol. 6, pp. 163–180, 1985.

[6] B. Jeon, J. Park, and J. Jeong, "Huffman coding of DCT coefficients using dynamic codeword assignment and adaptive codebook selection," *Signal Processing; Image Commun.*, vol. 12, pp. 253–262, 1998.

[7] K.-Y. Yoo, J.-D. Kim, and Y.-L. Lee, "A local statistics adaptive entropy coding method for the improvement of H.26L VLC coding," *Proc. SPIE*, vol. 4067, pp. 56–63, 2000.

[8] W. H. Chen and W. Pratt, "Sense adaptive coder," *IEEE Trans. Comm.*, vol. COM-32, pp. 225–231, 1984.

[9] Independent JPEG's Group JPEG Software, Release 4 (1992). [Online]. Available: http://www.ijg.org/

[10] M. Nelson and J. Gailly, *Data Compression Book*, 2nd ed. New York: M&T Books, 1996.

[11] A. Moffat, R. Neal, and I. Witten, "Arithmetic coding revisited," in *Proc. Data Compression Conf.*, Mar. 1995, pp. 202–211.

[12] N. Memon, "Adaptive coding of DCT coefficients by Golomb-Rice codes," in *Proc. 1998 Int. Conf. Image Proc.*, 1998, pp. 516–520.