Lossless Image Compression having Compression Ratio Higher than JPEG

Madan Singh
madan.phdce@gmail.com,
Vishal Chaudhary
Computer Science and Engineering, Jaipur National University, Jaipur Rajasthan, India
vishalhim@yahoo.com

----------ABSTRACT----------
This paper focuses on improving the Image compression ratio and having 100% quality factor. The major emphasis has been on Arithmetic Coding, its principle has been applied to get the wanted results i.e. higher compression ratio than JPEG on an average quality factor. JPEG algorithm is context dependent it is most valuable when there is less variation in image pixels values, but still JPEG is mostly used image compression standard.

Keywords—**JPEG, Arithmetic Coding, Recursion.**

I. INTRODUCTION

Images provide much easier way to interpret something more frequently than text, in todays' world images are being used accessibly as a form of communication, internet acting as main carrier. Like other things in the world it needs space know as memory space for its existence, space occupied by an image is known as size of image, and large file size has following consequences: Memory Requirement is more, Transmission speed is less, Take more time for read & write operation. So image compression do becomes necessary in today's world of high speed multimedia, where user's convenience is more necessary along quick response time of computer system. With time no. of image compression schemes has been proposed but the one that has dominated the world is JPEG. Efforts has been going on to further improve JPEG algorithms, this paper gives an algorithm that tends to give better result than JPEG without having any loss in the image data.

A. Data Compression

It is the process of reducing file size either by preserving its original information or by losing some of its original information [2].

B. Lossless Compression

Lossless compression techniques, as their name implies, involve no loss of information [7]. If data have been loss-lessly compressed, the original data can be recovered exactly from the compressed data. Lossless compression is generally used for application that can't tolerate any difference between original and reconstructed data [2].

C. Lossy Compression

Lossy compression techniques involve some loss of information, and that have been compressed using lossy techniques generally can't recovered exactly. Generally higher compression ratios are achieved with this compression rather than lossless compression. In many applications, this lack of exact reconstruction is not a problem, Image compression is an application of it [2].

II. TECHNIQUES

A. *Arithmetic Coding*

The basic fundamental behind this scheme is to generate a unique identifier for a symbol or sequences of symbols [2]. Such unique identifier is known as tag, one possible set of tags for representing sequences of symbols are the numbers in the interval [0, 1). Because the number of numbers in this interval is infinite, it should be possible to assign a unique tag to each distinct sequence of symbols [5].

1) *Generation of Tag*

The given unit interval (UI) [0, 1), for generating unique tag for each distinct sequence of symbol, the concept of cumulative sum is used, for given n no. of symbols (included repeated), any value is given each symbol or sequence in such a way that cumulative sum has to be exactly 1 and 0 value is not assigned to any one or the simple way to assign such value is to the concept of probability as used in case of Huffman coding [4]. Let F (x_1), F (x_2), F (x_3) be the values assigned to first, second and third character of sequence (x_1, x_2, x_3,……x_n), so in general way for n is the length of sequences of characters, F (x_i) for i = 1,2,3……..n can represent the values. Now divide the UI in n numbers of sub intervals, each interval have its symbol assigned, and interval of any symbol [F(i-1), F(i)).For any character select its corresponding interval and divide that into again n no. of sub- intervals by defining new sub – intervals values to each symbol , these sub-intervals values are defined for sequence (x_1, x_2, x_3,……x_n) are defined by following formulas:

$$l^n = l^{n-1} + (u^{n-1} - l^{n-1})F(x_{n-1})$$

$$u^n = l^{n-1} + (u^{n-1} - l^{n-1})F(x_n)$$

Here l^n minimum value of n_{th} sub interval u^n is the maximum value that same interval. Midpoint of interval is used as tag once last symbol of a given sequences is processed, then tag value is given by [4]:

$$T(X) = (u^n + l^n)/2$$

2) *Deciphering a Tag*

Decoding the tag is almost same process similar to above encoding process. Select the first symbol find apply encoding procedure on first interval if condition $l^1 <= T(x) < u^1$ then decoded character is first of sequence, if the given character is not of 1_{st} interval then same procedure is applied on next intervals until condition get satisfied [5]. Once a character is get decoded new l^n & u^n are used to as reference for rest of symbol's decoding.

3) *Transform coding*

Transform coding involves rather than using input sequences of pixel's values, the concept of variance in input sequence is used. This can be achieved by transforming sample's values into another sample type. In order to understand transform coding, go to the concept of derivatives. Given velocity time graph defined by v =f (t), then derivative of v is v` = f`(t), this v` = f`(t), tells about rate of change of velocity w.r.t parameter t. The derivative graph gives the indication of frequency of distribution or variability in the sample. This property is exploited for lossy image compression. There two kinds of transforms discrete and continuous. Discrete transform is of our importance as pixel's values are of discrete form. The commonly used discrete transform is Discrete Cosine Transform (DCT).

1. DCT

The Discrete Cosine Transform (DCT) gets its name from the fact that rows of N * N Transform Matrix A are obtained as a functions of cosines.

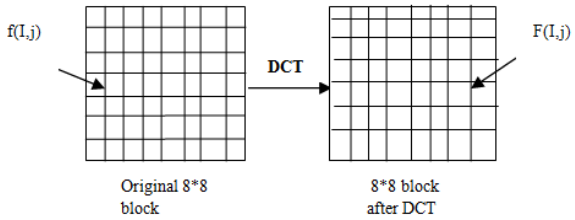

Fig.1 DCT conversion

$$F(u,v) = \left(\frac{2}{N}\right)\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}\lambda(i)\lambda(j)\cos\left[\frac{\pi u(2i+1)}{2N}\right]\cos[\pi v(2j+1)]\frac{f(i,j)}{N}$$

$$\lambda(\epsilon) = \begin{cases} \frac{1}{\sqrt{2}} & for\ \mu = 0 \\ 1 & otherwise \end{cases}$$

DCT gives the indication of variability in the sample with DCT co-efficient (F(i,j)). These value of DCT co-efficient are used to get approximate value of pixels with help of inverse DCT (IDCT) as follows:

$$F(i,j) = \left(\frac{2}{N}\right)\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}\lambda(u)\lambda(v)\cos\left[\frac{\pi i(2u+1)}{2N}\right]\cos[\pi j(2v+1)]\frac{f(u,v)}{N}$$

$$\lambda(\epsilon) = \begin{cases} \frac{1}{\sqrt{2}} & for\ \mu = 0 \\ 1 & otherwise \end{cases}$$

2. JPEG

The JPEG standard is one of the most commonly used standards for lossy image compression. The approach recommended by JPEG is a transform coding approach using the DCT.

The input image is first "level shifted" by 2^n-1 i.e. subtract 2^n-1 from each pixel value, where n is no. bits used to represent each pixel. Thus, for 8-bit images whose pixels take on values between 0 and 255, subtratct 128 from each pixel so that the value of pixel varies between -128 and 127. The image is divided in blocks of 8*8, which are then transformed using an 8*8 forward DCT. If any dimension of the image is not multiple of eight, the encoder replicates the last column or row until the final size is a multiple of eight. These additional rows or columns are removed during the decoding process. After applying DCT, DCT coefficients are obtained such that the lower frequency coefficients are in the top left corner of 8*8 block have larger value than the higher frequency components . This is the generally the case, Except for situations in which there is substantial activity in the image block.

- *Quantisation in JPEG*

The JPEG algorithm uses uniform quantisation to quantize the various coefficients. The quantizer step size are organized in a table called *quantisation table* and can be viewed as fixed part of quantisation. Each quantized value is represented by a label. The label corresponding to quantized value of the transform coefficient DCT_{ij} is obtained as

$$l_{ij} = \text{floor}\ ((DCT_{ij} / Q_{ij})+0.5)$$

Q_{ij} is the quantisation value taken from the *sample quantisation table* given below.

- *Sample Quantisation Table*

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	194	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

The sample quantisation table shows that the step size generally increases as moving from top left corner (DC coefficient) to the higher order coefficient. Because the quantisation error is an increasing function of step size, more quantisation error will be introduced in higher frequency coefficients than lower frequency coefficients. The decision on the relative size of step sizes is based on how errors in these coefficients will be perceived by human visual system. Quantisation errors in the DC and lower AC (except top left corner of 8*8 DCT coefficients block) coefficients are more easily detectable than the quantisation error in AC coefficients. Therefore, larger step sizes are used for perceptually less important coefficients. All the

coefficients with magnitudes less than half of step size will be set to zero. Because the step sizes at the tail end of the scan are larger, the probability of finding a long run of zeros increases at the end of zig-zag scan.

- *Zig-Zag Scan*

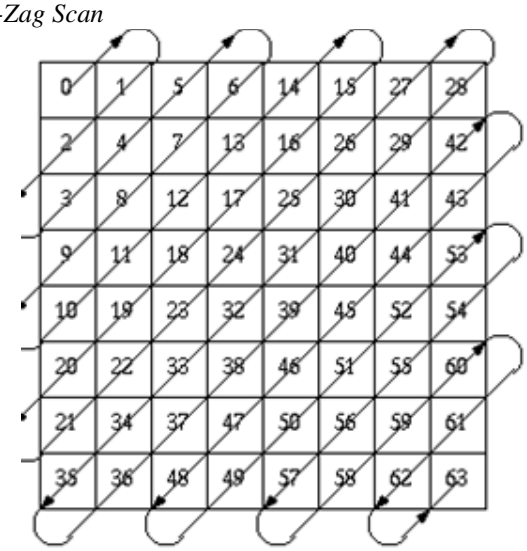

Fig.2 (zig zag scan)

- *Decoding of 8*8 block*
  To obtain a reconstruction of the original block, perform the de-quantization, which simply consists of multiplying the labels with corresponding value in *sample quantitation table*. Taking the inverse DCT transform of quantized coefficients and adding 128, will result in reconstructed block. In practical reproduction is remarkably close to original. For even more accurate reproduction, the step sizes in quantitation table multiplied by one half and using these values as new step sizes. This will result in a increment in bit rate at the cost of increased quality .

B. ***Recursive Merging***

This method is simple as used in merge sort i.e. first divide the image into two sub matrices that are further divided into two sub matrices until each sub matrix in of 8*8 (Leaf node). After applying any operation on those leaf nodes, information obtained is referred to as a record i.e. recombined with the help backtracking principle (recursion). The final information of image is an array of records.

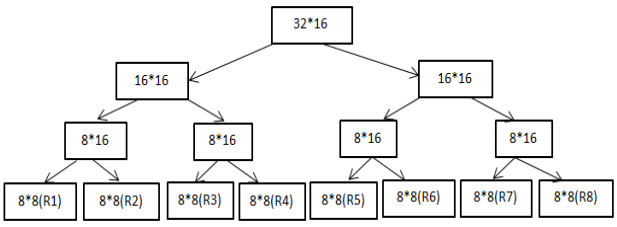

Fig.3 Decomposition of Image

R_n refers to record no. n in the array, this array is final information of image i.e. is going to be used to reconstruct the image. The size of this array is final size of image after compression.

III. IMPLEMENTATION

Following are the steps of Lossless Image compression with arithmetic coding (LICAC): **Encoding**

1. Divide the image into 8*8 blocks as described in II.C.
2. Covert all the pixels values into their respective binary values of 8-bit binary code (as all the values of gray scale image going to be covered by 8 -bit).
3. Apply arithmetic coding on all 64 * 8 (512) binary values, total no. of different characters is two i.e. 0, 1.
4. Output of arithmetic coding is a real value, this value and occurrence of 1 or 0 constitutes a record.
5. goto step (2) until all the remaining blocks are not processed, it results in the formation of array of records.
6. Convert the values of records into their equivalents binary values, here a record constitutes of bits representing arithmetic code value and an integer value.
7. An array of records is treated as a stream of binary values 0s & 1s.
8. Divide stream into small stream of size 512.
9. Apply arithmetic coding on the small streams, resulting in new arithmetic codes.
10. Make a new array where each entry is made up of new arithmetic codes and their respective occurrence of either 0s or 1s.
11. New array is the final array i.e. going to be used while decoding

Algorithm for LICAC: Data structure used {IMAGE a N*M matrix, Record [an array of type struct (double,int)]}

```
Record LICAC (IMAGE , N , M) {
String s;
for i = 1 to n
 for j = 1 to n
  S += (String)binary (IMAGE [i,j]);
/* Binary value of pixel is added to string */
int ones = count_no_1s (S);
/* counts no. of 1s in the String s & return int value */
  Zeros = 512-ones;
  /* no. of occurrence of 0s.*/
  Double code;
  code=Arithmetic_Encoding(s,ones,zeros)
  Record = {code,ones};
  /* Record consists of code and int  value.*/
  return Record;
  }
```

Following function uses the principle of divide and conquer to get the final result this uses following new terms in addition to previous ones lbr, lbc, upr, upc lower bound for rows,cloumns and upper bound for rows and columns respectively.

```
MERGE
(IMAGE,N,M,Record,lbr,lbc,upr,upc){
 static int i =1;
 p = floor( (lbr + upr)/2)
 q = floor( (lbc + upc)/2)
 if (N >= M && ((N/2) && M)!= 8)
 {
```

```
MERGE(IMAGE,N/2,M,Record,lbr,lbc,p,upc);

MERGE(IMAGE,N,M/2,Record,lbr,lbc,upr,q);
   Record [i++] = LICAC (IMAGE,N,M);
  }                              }
```

Now following procedure encodes the Record structure array. Algorithm for this procedure uses Data structures { Record [an array of type struct (double,int)], FinalRecord [an array of type struct (double,int)]}.

```
ProcessRecord (Record , FinalRecord) {
String S,Temp;
  for i = 1 to n
    S += (String)binary (Record[i]);
   /* Binary value of Record Strructure is added to string */
  for i = 1, j = 1 to S.Length{
     Temp = smallStream(S);
  /*Binary string is divided into small stream of string of length 512*/
          i = i + 512;
         int ones = count_no_1s (s);
       /* counts no. of 1s in the String s & return int value */
         Zeros = 512-ones;
       /* no. of occurrence of 0s.*/
         Double code;
        code=Arithmetic_Encoding
        (Temp,ones,zeros)
         FinalRecord[j++] = {code,ones};
      /* Record consists of code and int  values.*/
     }
  }
```

Size of FinalRecord structure array is final size of image, this structure is only i.e. going to be used for re-construction of image.

Following **Decoding** algorithm describes this process.

1. Construct Record structure array from FinalRecord structure array.
2. Apply recursive procedure to in order to get 8*8 blocks.
3. Use Arithmetic decoding procedure to get 8*8 image blocks.
4. Recursive merging is used to get the original image.

Following algorithm construct Record structure array. Algorithm for this functions uses Data structures { Record [an array of type struct (double,int)], FinalRecord [an array of type struct (double,int)]}.

```
ReconstructionRecordArray (Record ,
FinalRecord)
{
  len = FinalRecord.length;
  /*Length of record array */
  for i = 1 to len{
      int n =1;
    String s = Arithmetic_Decoding
    (FinalRecord[i]);
  /*Arithmetic decoding is applied to generate the binary string encoded  during
compression*/
     int count = 1;
      for j = 1 to 512{
       Record [j]=
decimal(s.substring(n,n+80));
       /* groups of 80 bits binary string here first 64 bits represent
   arithmetic code value last 16 bits represent the occurrence of either  0s or 1s
   that are  converted to their respected decimal values, all this constitutes a
   Record */
       If (count++ = 6) break;
     /* 6 Record elements a FinalRecord element /*
       }
   }
  }
```

Data structure required for this process is Heap, total no. of entries in heap is = 2*Record.length -1, leaf nodes entries of the heap is going to be as of record, the constructs the internal nodes of the heaps. The root of the heap is final Image. Heap data structure contains items of type Matrix. Following algorithm uses following items ,Heap ,IMAGE, Record.

```
Reconstruction (IMAGE,Record,Heap )
{
  len = Record.length;
  /*Length of record array */
  int Matrix [8*8];
  /* used for storing temp. 8*8 block */
  for i = 1 to len {
    int n =1;
   /* used for making 8-bit group */
   String s = Arithmetic_Decoding
(Record[i]);
  /*Arithmetic decoding is applied to generate the binary string encoded during
  compression*/
   for j =1 to 8
     for k = 1 to 8
      Matrix
[j,k]=decimal(s.substring(n,n+8));
   /* groups of 8 bits binary string i.e. converted to decimal value */
       n += 8;
       Heap [len + i-1] = Matrix
  }
  i = 1; len = Heap.length;
  while len != 1 {
   Heap[len/2]=Heap[len + i-1]+Heap[len
-i];
   len -= 2;}
  IMAGE = Heap[1]; // final Image              }
```

IV. RESULTS

Consider a sequence of alternative 1s ,0s constructing a sequence of length 512, both have probability of 0.5 required for arithmetic coding. Following is the sequence of real values is going to be generated represented in the form of generating function.

0.5 + 0.25 + 0.125 + 0.03125 + 0.015625 + 0.00078125 + …….. + 7.458340731200207E-155

This is a GP series having r = 0.5
T_{512} = a * r^{511} = 0.5^512 = 7.458340731200207E-155
T_{512} is maximum value i.e. going to be achieved during generation of code during Arithmetic encoding, amount of memory required for storing this value is 8 bytes. As described earlier in section II that 2 bytes are also required for storing the values of either 1 or 0. So total bytes required for a 8*8 block is 10 bytes.

Following is the table for some random images, their quality factors (QF) are processed using Adobe Photoshop.

Table1. JPEG vs LICAC

Original Size (KB)	JPEG Size (KB) (Low QF)	JPEG Size (KB) (Medium QF)	JPEG Size (KB) (Maximum (QF)	LICAC Size(KB)
97.5	32.9	44.2	81.4	2.54
253	46.2	69.2	145.8	6.6
601.8	101	118.6	331.3	15.68
698.9	122.5	143.9	400.5	18.3
812.3	144.3	163.2	426.6	21.2
1160	183.0	219.7	646.1	30.3
1530	250.3	290.6	868.1	39.9
2050	325.1	355.3	1000	53.4
2130	335.7	371.6	1005	55.5
3000	475.8	517.6	1024	78.13
Average	**201.7**	**229.7**	**592.8**	**32.12**

Table2. % Performance Increased by LICAC

JPEG (QF)	% Increase (100% Quality)
Low	84
Medium	86
Maximum	94.5

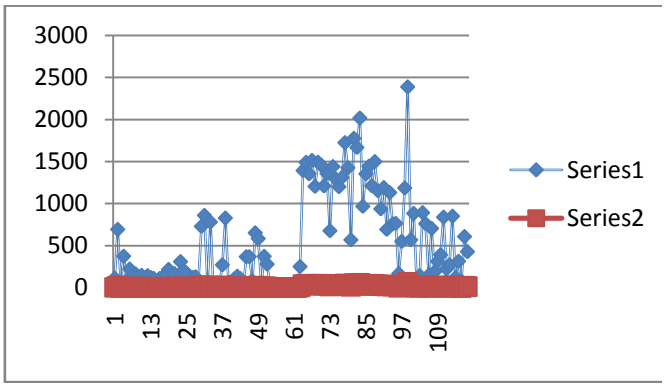

Fig.4 Series1 (JPEG) vs Series2 (LICAC)

Above fig. shows the results for size of file after JPEG compression and size of files after LICAC compression. These results are based upon random sample of sizes of various images.

V. CONCLUSION

JPEG image compression standard gives poor result when quality is more important as compared to LICAC. Even on medium quality JPEG performance is 86% less than LICAC. JPEG suffers big time when original image is less than 200 KB. LICAC compression algorithm gives 100 % image quality still having 84% compression performance better than JPEG with low quality factor. As depicted by Table1 that LICAC's performance is 94.5% better than JPEG still having no loss in the image quality. On the LICAC's performance is 84% better than JPEG considering average case of (201.7, 229.7, 592.8) table1 with pure quality.

VI. REFERENCES

[1] J,R, Pierce, *Symbols, Signal, and Noise-The Nature and process of communications*, Harper, 1961.
[2] T,C Bell, J,G, Clearly, and I,H Witten, *Text Compression, Advanced Reference Series*, Prentice Hall, Englewood Cliffs, NJ, 1990.
[3] S, Pigeon, *Huffman Coding, In K, Sayood*, editor, Lossless Compression Handbook, Pages 79-100, Academic, 2003.
[4] A, Said, *Arithmetic Coding, In K sayood*, editor, lossless Compression Handbook, pages 101-152, Academic Press 2003. .
[5] Khalid Sayood I*ntroduction to Data Compression*, Third edition, Elsevier, 2011.
[6] W,B, Pennebaker and J,L, Mitchell, *JPEG still image compression Standard,* Van Nostrand Reinhold, 1993.
[7] D. Salomon, *Data Compression*: *The Complete Reference*, Springer, 1998.