Lossless Image Compression and Transmission

A Special Assignment Report

Submitted in the
Open Elective Course

FUNDAMENTALS OF IMAGE AND VIDEO PROCESSING

By

Ayush Makwana, Nihar Markana
18BCE107, 18BCE116
B.Tech Sem V
Computer Science and Engineering

Electronics and Communication Engineering Department
Institute of Technology, Nirma University
Ahmedabad-382481
November 2020

ABSTRACT

To store large-sized images and to make them available on the internet, compression techniques are needed. Image compression addresses the problem of reducing the amount of data required to represent a digital image. The underlying basis of the reduction process is the removal of redundant data. According to a mathematical point of view, this amounts to transforming a two-dimensional pixel array into a statistically uncorrelated data set. At the receiver, the compressed image is decompressed to the approximation of the original image. There are two types of compression techniques, lossless compression, and lossy compression. In general, lossy compression techniques have the advantage of greater compression ratios and a disadvantage of image quality degradation. Huffman coding, a lossless image compression technique is implemented in the following sections.

1) Introduction

1.1 Problem Statement

For understanding a problem statement let's take an example: An image, 1024 pixel x 1024 pixel x 24 bit, without compression, would require 3 MB of storage and 7 minutes for transmission, utilizing a high speed, 64 Kbit/s, ISDN line. If the image is compressed at a 10:1 compression ratio, the storage requirement is reduced to 300 KB and the transmission time drops to under 6 seconds. Hence, Compression is an important component of the solutions available for creating file sizes of manageable and transmittable dimensions. Increasing the bandwidth is another method, but the cost sometimes makes this a less attractive solution. Platform portability and performance are important in the selection of the compression/decompression technique to be employed. The easiest way to reduce the size of the image file is to reduce the size of the image itself. By shrinking the size of the image, fewer pixels need to be stored and consequently, the file will take less time to load. and also by making lossless image compression we compressed the image and transmitted it without any type of loss.

1.2 Objective

Image compression plays an impassive role in memory storage while getting a good quality compressed image. The objective of image compression is to reduce the irrelevance and redundancy of the image data to be able to store or transmit data in an efficient form. Image compression may be lossy or lossless The goal of lossless image compression is to represent an image signal with the smallest possible number of bits without loss of any information, thereby speeding up transmission and minimizing storage requirements.

2) Literature Review / Description

Huffman Coding

Huffman's code procedure is based on the two observations.

a. More frequently occurred symbols will have shorter code words than symbols that occur less frequently.

b. The two symbols that occur least frequently will have the same length.

Table 1: Huffman source reduction.

Original source		Source reduction			
S	P	1	2	3	4
a2	0.4	0.4	0.4	0.4	0.6
a6	0.3	0.3	0.3	0.3	0.4
a1	0.1	0.1	0.2	0.3	
a4	0.1	0.1	0.1		
a3	0.06	0.1			
a5	0.04				

S-source, P-probability

Table 2 : Huffman Code Assignment Procedure

Original source Source reduction

S	P	1	2	3	4
a2	0.4[1]	0.4[1]	0.4[1]	0.4[1]	0.6[0]
a6	0.3[00]	0.3[00]	0.3[00]	0.3[00]	0.4[1]
a1	0.1[011]	0.1[011]	0.2[010]	0.3[01]	
a4	0.1[0100]	0.1[0100]	0.1[011]		
a3	0.06[01010]	0.1[0101]			
a5	0.04[01011]				

S-source, P-probability

At the far left of the table I the symbols are listed and corresponding symbol probabilities are arranged in decreasing order and now the least two probabilities are merged as here 0.06 and 0.04 are merged, this gives a compound symbol with probability 0.1, and the compound symbol probability is placed in source reduction column1 such that again the probabilities should be in decreasing order. so this process is continued until only two probabilities are left at the far right shown in the above table as 0.6 and 0.4.

The second step in Huffman's procedure is to code each reduced source, starting with the smallest source and working back to its original source. The minimal length binary code for a two-symbol source, of course, is the symbols 0 and 1. As shown in table II these symbols are assigned to the two symbols on the right (the assignment is arbitrary; reversing the order of the 0 and would work just and well).

As the reduced source symbol with probabilities, 0.6 was generated by combining two symbols in the reduced source to, the 0 used to code it is now assigned to both of these symbols, and a 0 and 1 are arbitrarily appended to each to distinguish them from each other. This operation is then repeated for each reduced source until the original course is reached.

The final code appears at the far-left in the table. The average length of the code is given by the average of the product of the probability of the symbol and the number of bits used to encode it. This is calculated below

Lavg = (0.4)(1) +(0.3)(2) + (0.1)(3) + (0.1)(4) + (0.06)(5) + (0.04)(5) = 2.2 bits/ symbol

and the **entropy of the source is 2.14bits/symbol**,

the resulting **Huffman code efficiency is2.14/2.2 = 0.973.**

Entropy, H=-ΣP(aj)log P(aj)

Huffman's procedure creates the optimal code for a set of symbols and probabilities subject to the constraint that the symbols be coded one at a time.

Huffman Decoding

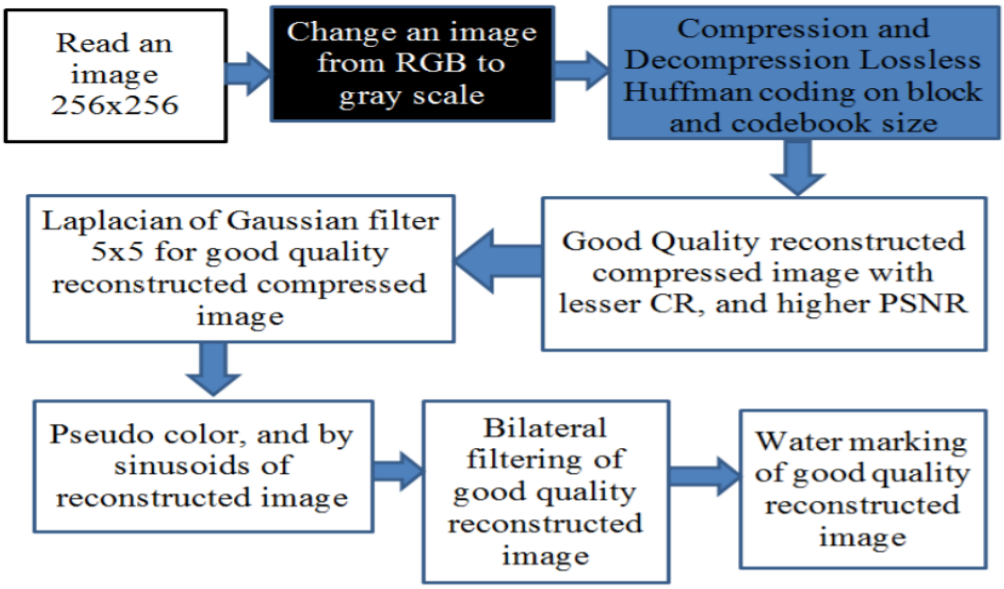
After the code has been created, coding and/or decoding is accomplished in a simple look-up table manner. The code itself is an instantaneous uniquely decodable block code. It is called a block code because each source symbol is mapped into a fixed sequence of code symbols. It is instantaneous because each codeword in a string of code symbols can be decoded without referencing succeeding symbols. It is uniquely decodable because any string of code symbols can be decoded in only one way. Thus, any string of Huffman encoded symbols can be decoded by examining the individual symbols of the string in a left to right manner.

For the binary code of table 2, a left-to-right scan of the coded string **010100111100** reveals that the first valid codeword is **01010**, which is the code for symbol **a3**. The next valid code is **011**, which corresponds to symbol **a1**. Valid code for the symbol **a2** is **1**, valid code for the symbols **a6** is **00**, valid code for the symbol **a6** is Continuing in this manner reveals the completely decoded message a5 a2 a6 a4 a3 a1, so in this manner, the original image or data can be decompressed using Huffman decoding as explained above.

At first, we have as much as the compressor does a probability distribution. The compressor made a code table. The decompressor doesn't use this method though. It instead keeps the whole Huffman binary tree, and of course a pointer to the root to do the recursion process. In our implementation, we'll make the tree as usual and then you'll store a pointer to the last node in the list, which is the root. Then the process can start. We'll navigate the tree by using the pointers to the children that each node has. This process is done by a recursive function that accepts as a parameter a pointer to the current node and returns the symbol.

3 Methodology

Step 1: Reading MATLAB image.
Step 2: Converting RGB image to Gray-scale level image
Step 3: Call a function that finds the symbols for the image
Step 4: Call a function that calculates the probability of each symbol for image
Step 5: The probability of symbols should be arranged in DESCENDING order so that the lower probabilities are merged. It is continued until it is deleted from the list and replaced with an auxiliary symbol to represent the two original symbols.
Step 6: In this step, the code words are achieved related to the corresponding symbols that result in a compressed data/image.
Step 7: Huffman code words and final encoded Values (compressed data) all are to be concatenated.
Step 8: Huffman codewords are achieved by using final encoding values. This may require more space than just the frequencies that is also possible to write the Huffman tree on the output
Step 9: Original image is reconstructed in the spatial domain which is compressed and/or decompression is done by using Huffman decoding.
Step 10: Compressed image applied on Huffman coding to get the better quality image based on block and codebook size.
Step 11: Recovered reconstructed looks similar to the original image.
Step 12: Implement Laplacian of Gaussian 5x5 filtering for lossless Huffman coding compressed image
Step 13: Implement Pseudo coloring for lossless Huffman coding compressed image
Step 14: Implement Bilateral filtering for lossless Huffman coding compressed image
Step 15: Implement Watermarking for lossless Huffman coding compressed image



4) Implementation Results

```
>>file1


      Variance : 13

      Average Length :7.865064e+00

Time Elapsed :    4.3565
```

```
>>file2
16  title('Image after Transmission');
```

```
      Entropy is        :7.840854 bits

      Efficiency is     :0.996922
```

5) Conclusion and Future Scope

The future scope is the visibility of lossless Huffman coding to use in other advanced image enhancement techniques.

Efficient and Effective communication of superior quality digital images needs a reduction of memory space and less bandwidth requirement.
(a) Good quality image with Lower compression ratio.
(e) Lower entropy and more the Average Length.

Appendix

Probability.m file :

```
function [freq,probability,symbol] = prob(r)
     %Prob function calculate the
     %     frequency-> freq ;
     %     Probability ->probability
     %     symbol present in Image between 0-255
     %     Omitting symbols which are not present
     %% Extracting the Dimensions of Image
     [lenx,leny,lenz]=size(r);
     %% For Loop going through every pixel of Image
     %  calculating Frequency of symbols

     freq=zeros(256,1);
     for b=1:lenx
        for j=1:leny
            for k=1:lenz
            val=r(b,j,k);
            freq(val+1,1)=freq(val+1,1) +1;
            end
        end
     end
     %% Here we count Number of symbols not appearing in Image

     count=0;
     f=freq';
     for k=1:256
        if (f(k)==0)
            count=count+1;
        end
     end
     %% Generating present symbol File in Ascending order
```

```
            n=256-count;
            symbol=zeros(n,1);
            b=1;
            for k=1:256
               if (f(k)==0)
               else
                    symbol(b)=k;
                    b=b+1;
               end
            end
            %% Probability calculation of every symbol in Image

            probability=zeros(n,1);
            for k=1:n
               probability(k)=freq(symbol(k));
            end
            probability=probability/(lenx*leny*lenz);
      end
      %%Symbol Vector and Probability vector are of same size.
      % frequency vector is of length 256
```

PriorityQueue.m file:

```
% A simple priority queue that takes arrays as inputs and uses a
specific
% column of the 1xN arrays to sort by the minimum value.
%
% Utilizes a minheap to ensure quick operation even with queues with
% > 100,000 elements
classdef PriorityQueue < handle
   properties (Access = private)
       Data % the data in the queue
       Size % the size of the queue
       Column % the comparator column of the data
   end
   methods
       % contstuctor of the queue
       function obj = PriorityQueue(varargin)
           obj.Size = 0;
           obj.Data = {};
           if size(varargin) == 1
               obj.Column = varargin{1};
           else
               obj.Column = 1; % default comparator column is the
first
```

```
            end
        end

        % insert an array into the queue given by arg1
        function insert(obj, arg1)
            if size(arg1,2) < obj.Column
                disp('error:insert array too small for queue');
                return
            end
            obj.Size = obj.Size + 1;
            obj.Data{obj.Size} = arg1; % add new element to bottom of
heap
            parentIter = floor(obj.Size/2);
            currentIter = obj.Size;
            % perform bubble up operation
            while parentIter > 0
                if obj.Data{currentIter}(obj.Column) <
obj.Data{parentIter}(obj.Column)
                    % the current element is smaller than parent so
swap
                    temp = obj.Data{currentIter};
                    obj.Data{currentIter} = obj.Data{parentIter};
                    obj.Data{parentIter} = temp;
                    currentIter = parentIter;
                    parentIter = floor(currentIter/2);
                else
                    break;
                end
            end
        end

        % remove and return the top array from queue or any that match
the input array
        function node = remove(obj, varargin)
            node = 0;
            if obj.Size == 0
                disp('remove from an empty queue, returing 0');
                return
            end
            if size(varargin,2) > 0
                for i=1:obj.Size
                    if isequal(obj.Data{i}, varargin{1})
                        node = obj.Data{i};
                        obj.Data{i} = [];
                        obj.Size = obj.Size - 1;
                    end
                end
```

```
                obj.Data = obj.Data(~cellfun('isempty',obj.Data)); %
remove empty element from cell array
            else
                node = obj.Data{1};
                obj.Data{1} = obj.Data{obj.Size}; % replace root with
last element
                obj.Data{obj.Size} = 0; % clear the last element
                obj.Size = obj.Size - 1;
                leftChild = 2;
                rightChild = 3;
                currentIter = 1;
                % perform bubble down
                while currentIter < obj.Size
                    if leftChild <= obj.Size && rightChild <= obj.Size
&& (obj.Data{currentIter}(1,obj.Column) >
obj.Data{leftChild}(1,obj.Column) ||
obj.Data{currentIter}(1,obj.Column) >
obj.Data{rightChild}(1,obj.Column))
                        if obj.Data{leftChild}(1,obj.Column) <
obj.Data{rightChild}(1,obj.Column)
                            % left child is smaller
                            tmp = obj.Data{currentIter};
                            obj.Data{currentIter} =
obj.Data{leftChild};
                            obj.Data{leftChild} = tmp;
                            currentIter = leftChild;
                            leftChild = currentIter*2;
                            rightChild = currentIter*2+1;
                        else
                            % right child is smaller
                            tmp = obj.Data{currentIter};
                            obj.Data{currentIter} =
obj.Data{rightChild};
                            obj.Data{rightChild} = tmp;
                            currentIter = rightChild;
                            leftChild = currentIter*2;
                            rightChild = currentIter*2+1;
                        end
                    elseif leftChild <= obj.Size && rightChild >
obj.Size && obj.Data{currentIter}(1,obj.Column) >
obj.Data{leftChild}(1,obj.Column)
                        % only the left child exists
                        tmp = obj.Data{currentIter};
                        obj.Data{currentIter} = obj.Data{leftChild};
                        obj.Data{leftChild} = tmp;
                        break;
                    else
```

```
                        % either the children are empty or the
currentIter value is minimum
                        break;
                    end
                end
            end
        end

        % peek and return the first element of the queue
        function node = peek(obj)
            if obj.Size == 0
                node = 0;
                disp('peek from an empty queue, returing 0');
                return
            end
            node = obj.Data{1}; % returns the first element in queue
        end

        % returns the size of the queue
        function sz = size(obj)
            sz = obj.Size; % returns the size of the queue
        end

        % clears the entire queue
        function clear(obj)
            obj.Data = {};
            obj.Size = 0;
        end

        % checks if the queue contains a specific array
        function flag = contains(obj, array)
            flag = 0;
            for i=1:obj.Size
                if isequal(obj.Data{i}, array)
                    flag = 1; % if the array exists in the queue
                    return
                end
            end
        end

        % returns all the elements of the queue as a cell array
        function queue = elements(obj)
            queue = obj.Data;
        end

    end
end
```

Huffman_Coding.m file:

```
function [dictionary,avglen] = huffman_encoding(symbol,p)
    m = length(symbol);
    dictionary = [];
    cache = [];
    for i=1:m
       dictionary = [dictionary;[{symbol(i)},{[]}]];
       cache = [cache;[{[symbol(i)]}]];
    end
    %map = containers.Map(p,linspace(1,m,m));
    map = [p linspace(1,m,m)'];
    index_map = zeros(256,1);
    for i=1:length(symbol)
       index_map(symbol(i)) = i;
    end
    count = 0;
    while(length(map))>1
       %disp(length(map));
       %fprintf('\n');
       count = count+1;
       map = sortrows(map);
       dict_one = cache{map(1,2)};
       dict_two = cache{map(2,2)};
       for i=1:length(cache{map(1,2)})
           %dictionary{index_map(dict_one(i)),2} =
    [dictionary{index_map(dict_one(i)),2} 1];
           dictionary{index_map(dict_one(i)),2} = [1
    dictionary{index_map(dict_one(i)),2}];
       end
       for i=1:length(cache{map(2,2)})
           %dictionary{index_map(dict_two(i)),2} =
    [dictionary{index_map(dict_two(i)),2} 0];
           dictionary{index_map(dict_two(i)),2} = [0
    dictionary{index_map(dict_two(i)),2}];
       end
       cache{map(2,2)} = [cache{map(2,2)} cache{map(1,2)}];
       map = [map;[map(1,1)+map(2,1) map(2,2)]];
       map = map(3:length(map),:);
       %break;
       if(length(map)==2)
           map = sortrows(map);
           dict_one = cache{map(1,2)};
           dict_two = cache{map(2,2)};
           for i=1:length(cache{map(1,2)})
               %dictionary{index_map(dict_one(i)),2} =
    [dictionary{index_map(dict_one(i)),2} 1];
```

```
                dictionary{index_map(dict_one(i)),2} = [1
        dictionary{index_map(dict_one(i)),2}];
            end
            for i=1:length(cache{map(2,2)})
                %dictionary{index_map(dict_two(i)),2} =
        [dictionary{index_map(dict_two(i)),2} 0];
                dictionary{index_map(dict_two(i)),2} = [0
        dictionary{index_map(dict_two(i)),2}];
            end
            cache{map(2,2)} = [cache{map(2,2)} cache{map(1,2)}];
            map = [map;[map(1,1)+map(2,1) map(2,2)]];
            map = map(3:length(map),:);
            break;
        end
    end
    len = zeros(length(symbol),1);
    for i=1:length(symbol)
        len(i) = length(dictionary{i,2});
    end
    avglen = sum(p.*len);
end
```

File1.m file:

```
%% Huffman Encoding
tic
%Clear all variables
clear;
%% Load image
image = imread('image2.jpg');
data = double(image);
dim=size(data);
%length of 1-Darray;
m = dim(1)*dim(2)*dim(3);
%reshape image into 1-D array
data = reshape(data,1,m);
%freq->frequency of each symbol(1-256)
%p->probability of each symbol
%symbol->array of symbols which are present(omit the symbols
which are not present)
[freq,p,symbol] = prob(data);
%% Huffman Encoding for the symbols present
%dict->dictionary which keeps record of symbols and
corresponding encoded bits
%avglen->average length of bits = sigma(frequency*probability)
%For image transmission without Huffman Encoding, Average Length
= 8bits
```

```
%[dict,avglen]=huffmandict(symbol,p);
%Our Algorithm
[dict,avglen]=huffman_encoding(symbol,p);
%Add '1' to adjuct indices according to Matlab default
data = data+1;
%Map existing symbols to dictionary index/key
map = zeros(256,1);
%Length Matrix
len = zeros(length(symbol),1);
for i=1:length(symbol)
   map(symbol(i)) = i;
   len(i) = length(dict{i,2});
end
%Extract only bits corresponding to each symbol in image
t = dict(map(data(:)),2);
%Concatenate all bits in row first order (Order
R,G,B,...R,G,B,....)
transmitted_bits = [t{:}];
%% Plot Original Image before Transmission
imshow(image);
title('Image before Transmission');
fprintf('\n\tVarience : %d',max(len)-min(len));
fprintf('\n\tAveragelength :%d \n',avglen);
%% Clear all other variables except
%   1.)dictionary->bits correspondin to symbol
%   2.)Bits/Information
%   3.)Dimension of image to extract from transmitted bits
%   4.)Average length of bits, probability(For the sake of
calculating entropy, efficiency)
%      ->Need not be transmitted through channel
clearvars -except dict transmitted_bits dim avglen p t
%clear map;
%Channel transmission -> Only 1,2,3 are transmitted through
channel
%run 'run_this_second.m' to extract lossless image from
tranmitted bits
%check
% for i=1:length(p)
%     sprintf('%d',dict{i,2})
% end
fprintf('Tima Elapsed :')
disp(toc);
```

File2.m file:

```
%% Huffman Decoding
```

```
%After transmission we decode the bits into symbols based in
dictionary
% and shape the image based on order of traversal
%Information is generally transmitted through Noise channel so
we calculate Entropy and Efficiency
%to tell how good this Huffman Lossless image compression and
transmission is
%% Extract symbols from transmitted bits
decoded_data=huffmandeco(transmitted_bits,dict);
%Reshape extracted data into image
extracted_image = reshape(decoded_data,dim(1),dim(2),dim(3));
%Extracted image after transmission
figure, imshow(uint8(extracted_image));
title('Image after Transmission');
%% Calculate Entropy and Efficiency
H=0;
n=length(p);
for k=1:n
   H=H+(p(k)*log2(1/p(k)));

end
fprintf('\n\tEntropy is\t\t:%f bits',H);
N=H/avglen;
fprintf('\n\tEfficiency is\t:%f\n',N);
```

References

[1]. International Research Journal of Engineering and Technology (IRJET). “Lossless Huffman coding image compression implementation in the spatial domain by using advanced enhancement techniques” Feb-2016

[2]. https://www.geeksforgeeks.org/image-compression-using-huffman-coding/

[3]. https://www.youtube.com/watch?v=7z2plqbm2gY

[4]. https://www.sciencedirect.com/topics/engineering/lossless-image-compression/