LOSSLESS IMAGE COMPRESSION

PROBLEM STATEMENT:

Image can be represented with minimum number of bits by using image compression. When images are transferred over the network it requires space for storage and time to transmit images. The present work investigates image compression using block truncation coding.

Three algorithms were selected namely, the original block truncation coding (BTC), Absolute Moment block truncation coding (AMBTC) and Huffman coding and a comparison was performed between these

Block truncation coding (BTC) and Absolute Moment block truncation coding (AMBTC) techniques rely on applying divided image into non overlapping blocks. They differ in the way of selecting the quantization level in order to remove redundancy.

In Huffman coding an input image is split into equal rows & columns and at final stage sum of all individual compressed images which not only provide better result but also the information content will be kept secure. It has been show that the image compression using Huffman coding provides better image quality than image compression using BTC and AMBTC. Moreover, the Huffman coding is quite faster compared to BTC

REQUIRED SOLUTION:

LOSSLESS IMAGE COMPRESSION

A simple task for Huffman coding is to encode images in a lossless manner. This is useful for precise and critical images such as medical images and Very High Resolution (VHR) satellite images, where it is important for the data to remain consistent before and after compression.

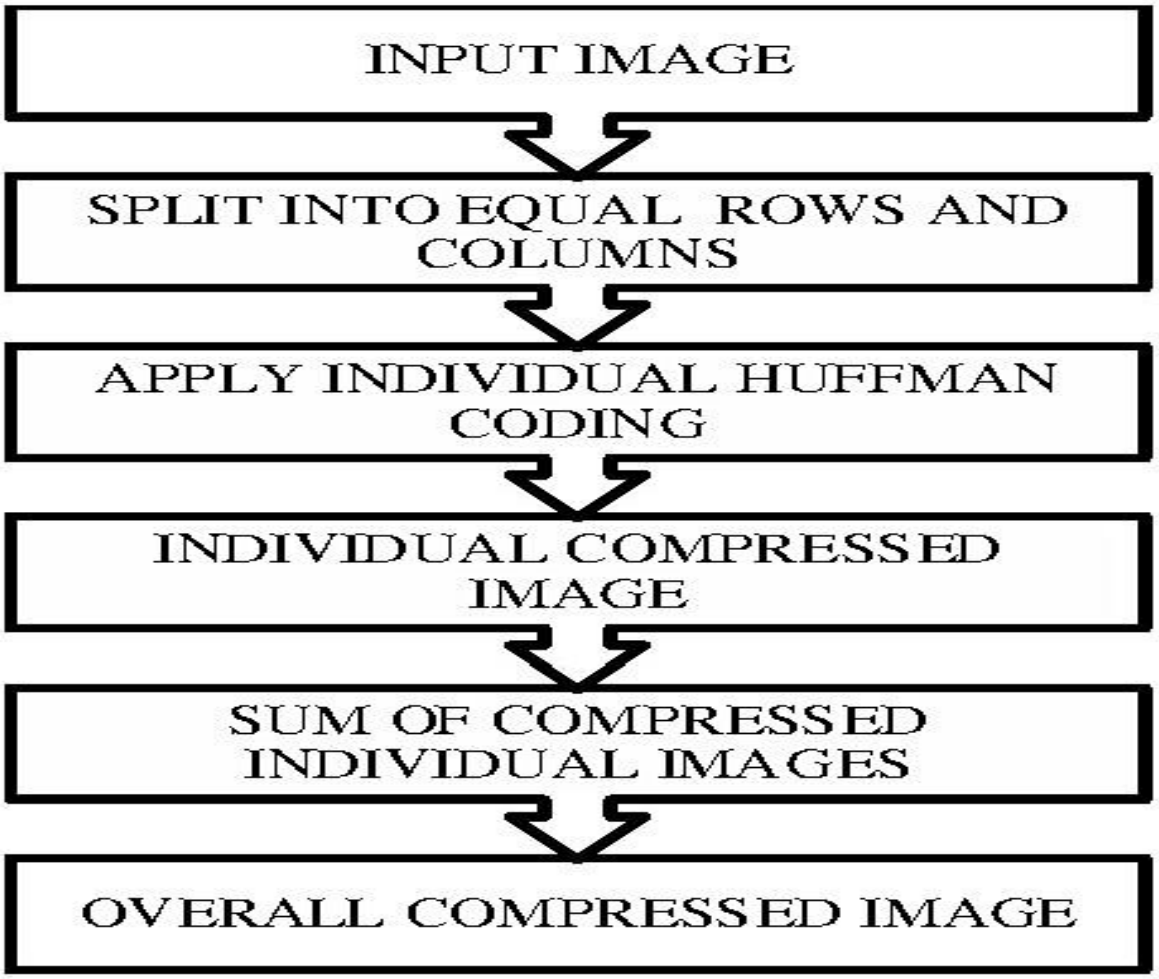
Most images can be represented as an *n* x *m* x *d* matrix of points where n is the height, m is the width, and d is the depth of the image. For grey scale images, each pixel has a depth of 1. For RGB (red-green-blue) images, each pixel has a depth of 3, one for each color. For multi spectral and hyper spectral images, each pixel can have a depth of over a hundred. For this article, only

RGB images will be used, but it is trivial to extend this to images with different depths.

Typically, the values that each number in the matrix can take on is an integer from 0 to 255. Encoding this range of numbers requires an 8-bit-number. The following section will show how Huffman coding can be used to compress images smaller than 8 bytes/point.

WORKING:

The lossy compression techniques lead to loss of data with higher compression ratio. Huffman coding is a loss less technique with more attractive features in various applications such as medical survey and analysis, technical drawing etc. Huffman coding has better characteristics of image compression. As we know that Huffman coding algorithm is a step by step process and involves the variable length codes to input characters & it is helpful in finding the entropy and probability of the state. It is effortless to calculate quality parameters in Huffman algorithm. Original image can be reconstructed with the help of digital image restoration



Step 1 :

Read the Image into a 2D array(**image**)

If the Image is in.**bmp** format, then the Image can be read into the 2 D array

Step 2 :

Define a struct which will contain the pixel intensity values (pix), their corresponding probabilities (freq), the pointer to the left (*left) and right (*right) child nodes and also the string array for the Huffman code word (code).

These struts is defined inside main(), so as to use the maximum length of code(maxcodelen) to declare the code array field of the struct pix freq

Step 3 :

Define another Struct which will contain the pixel intensity values(pix), their corresponding probabilities(freq) and an additional field, which will be used for storing the position of new generated nodes (arrloc).

Step 4 :

Declaring an array of structs. Each element of the array corresponds to a node in the Huffman Tree.

Step 5 :

Initialize the two arrays pix_freq and huffcodes with information of the leaf nodes

Step 6 :

Sorting the huffcodes array according to the probability of occurrence of the pixel intensity values

Note that, it is necessary to sort the huffcodes array, but not the pix_freq array, since we are already storing the location of the pixel values in the arrloc field of the huffcodes array

Step 7 :

Building the Huffman Tree

We start by combining the two nodes with lowest probabilities of occurrence and then replacing the two nodes by the new node. This process continues until we have a root node. The first parent node formed will be stored at index nodes in the array pix_freq and the subsequent parent nodes obtained will be stored at higher values of index.

Step 8:

Backtrack from the root to the leaf nodes to assign code words

Starting from the root, we assign '0' to the left child node and '1' to the right child node.

Now, since we were appending the newly formed nodes to the array pix_freq, hence it is expected that the root will be the last element of the array at index totalnodes-1. Hence, we start from the last index and iterate over the array, assigning code words to the left and right child nodes, till we reach the first parent node formed at index nodes. We don't iterate over the leaf nodes since those nodes has NULL pointers as their left and right child

Final Step :

Encode the Image

CODE:

```
 int i, j;
char filename[] = "Input_Image.bmp"; int data
= 0, offset, bpp = 0, width, height; long bmpsize
= 0, bmpdataoff = 0;
int** image;
int temp = 0;

// Reading the BMP File

FILE* image_file;
image_file = fopen(filename, "rb"); if
(image_file == NULL)
{    printf("Error Opening File!!"); exit(1);
}
else
{image = (int*)malloc(height * sizeof(int)); for (i =
        0; i < height; i++)
        {
    image[i] = (int*)malloc(width * sizeof(int));
        }
                int numbytes = (bmpsize - bmpdataoff) / 3;
   for (i = 0; i < height; i++)
        {    for (j = 0; j < width; j++) {fread(&temp, 3, 1,
                image_file); temp = temp & 0x0000FF;
                image[i][j] = temp;
```

```
                }}}
offset = 0; offset =
2;
fseek(image_file, offset, SEEK_SET); fread(&bmpsize, 4, 1,
image_file);
offset = 10;
fseek(image_file, offset, SEEK_SET);
fread(&bmpdataoff, 4, 1, image_file);
fseek(image_file, 18, SEEK_SET); fread(&width, 4,
1, image_file); fread(&height, 4, 1, image_file);
fseek(image_file, 2, SEEK_CUR); fread(&bpp, 2, 1,
image_file);
fseek(image_file, bmpdataoff, SEEK_SET);



// Building Huffman Tree

float sumprob; int
sumpix;
int n = 0, k = 0; int
nextnode = nodes; while
(n < nodes - 1)
{       sumprob = huffcodes[nodes - n - 1].freq + huffcodes
[nodes -n - 2].freq;
sumpix = huffcodes[nodes - n - 1].pix + huffcodes[nodes - n - 2].pix;
        pix_freq[nextnode].pix = sumpix; pix_freq[nextnode].freq = sumprob;
        pix_freq[nextnode].left = &pix_freq[huffcodes
[nodes - n - 2].arrloc];
```

```
pix_freq[nextnode].right = &pix_freq[huffcodes[nodes - n - 1].arrloc];
pix_freq[nextnode].code[0] = '\0'; i = 0;
while (sumprob <= huffcodes[i].freq) i++;
for (k = nnz; k >= 0; k--
{if (k == i)
{huffcodes[k].pix = sumpix;
 huffcodes[k].freq = sumprob;
 huffcodes[k].arrloc = nextnode;
}
else if (k > i)huffcodes[k] = huffcodes[k - 1];}n 1;nextnode+= 1;}

// Encode the Image

int pix_val;
// Writing the Huffman encoded
// Image into a text file
FILE* imagehuff = fopen("encoded_image.txt", "wb"); for (r = 0; r <
height; r++) for (c = 0; c < width; c++) { pix_val = image[r]; for (i = 0; i <
nodes; i++) if (pix_val == pix_freq[i].pix) fprintf(imagehuff, "%s",
pix_freq[i].code);}
fclose(imagehuff);

// Printing Huffman Codes printf("Huffmann
Codes::\n\n"); printf("pixel values -> Code\n\n");
for (i = 0; i < nodes; i++) { if (snprintf(NULL, 0, "%d", pix_freq[i].pix) == 2)
        printf("       %d -> %s\n", pix_freq[i].pix, pix_freq[i].code);
        else printf(" %d  -> %s\n", pix_freq[i].pix,pix_freq[i].code);}
```

TIME COMPLEXITY ANALYSIS:

Since Huffman coding uses min Heap data structure for implementing priority queue, the complexity is O(nlogn). This can be explained as follows-

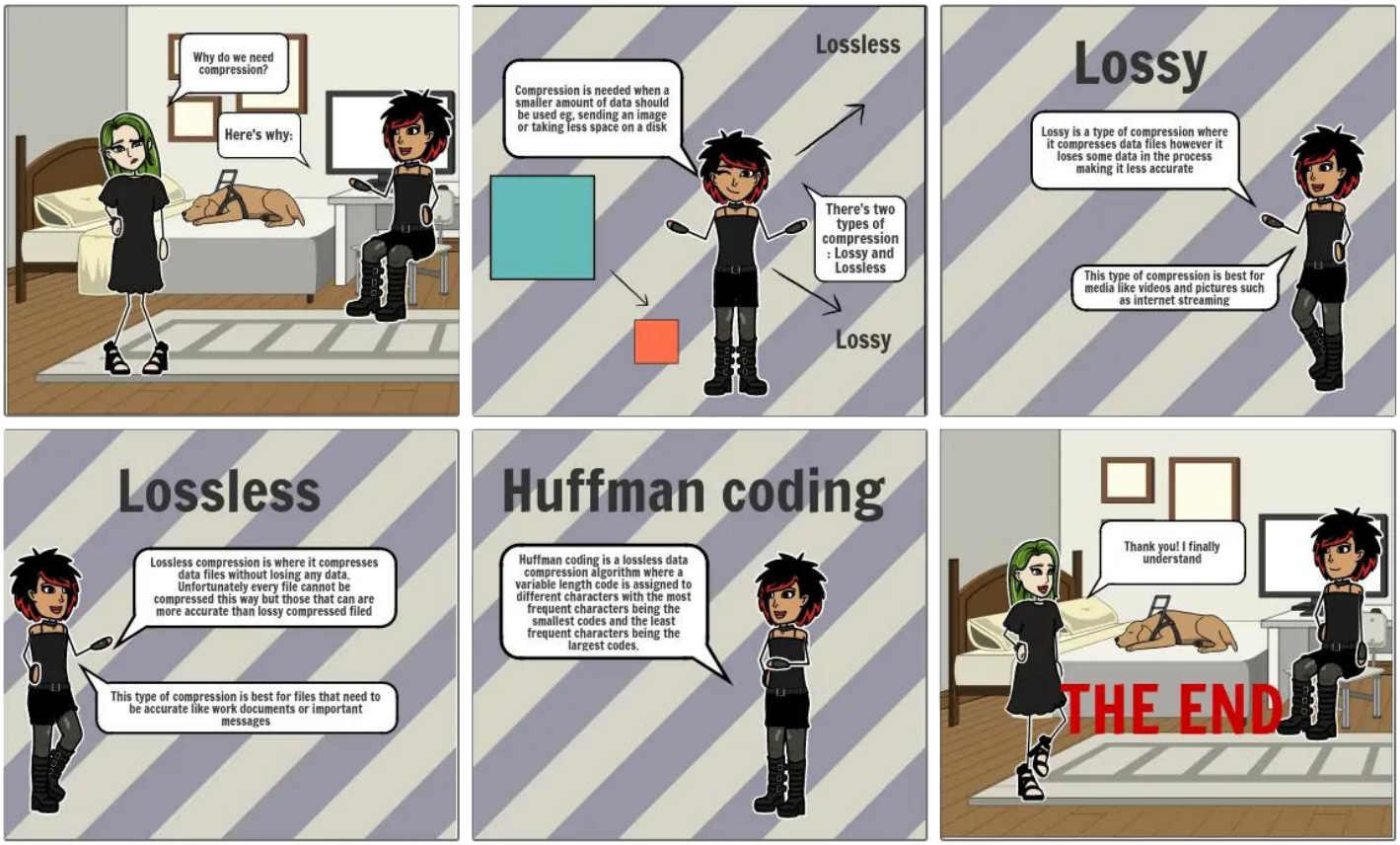
- **Building a min heap takes O(nlogn) time (Moving an element from root to leaf node requires O(logn) comparisons and this is done for n/2 elements, in the worst case).**
- **Building a min heap takes O(nlogn) time (Moving an element from root to leaf node requires O(logn) comparisons and this is done for n/2 elements, in the worst case).**

Since building a min heap and sorting it are executed in sequence, the algorithmic complexity of entire process computes to O(nlogn)

We can have a linear time algorithm as well, if the characters are already sorted according to their frequencies.

APPLICATIONS:

- **Huffman encoding is widely used in compression formats like GZIP, PKZIP (winzip) and BZIP2.**
- **Multimedia codecs like JPEG, PNG and MP3 uses Huffman encoding (to be more precised the prefix codes)**
- **Huffman encoding still dominates the compression industry since newer arithmetic and range coding schemes are avoided due to their patent issues.**



Create your own at Storyboard That

CONCLUSION:

Image compression plays vital role in saving memory storage space and saving time while transmission images over network.

Compression technique increase storage capability and transmission speed.Using compression coding techniques the shares size in image secret sharing is reduced. By using Huffman coding the image is compressed by 40%. Huffman algorithm is comparatively easier because of its simpler mathematical calculation in order to find the various parameters than BTC and AMBTC. Image compression plays vital role in saving memory storage space and saving time while transmission images over network. Compression technique increase storage capability and transmission speed.

TEAM MEMBERS

NAME	REG.NO
P SIVAMANI	RA2011026010261
JERAEMIUS SHANNON J	RA2011026010267
PL ARJUN MANIKANDAN	RA2011026010245