SaveDiskSpaceSaveEarth

CS 293 Final Project Demo

23rd November, 2012

Navin Chandak (#110050047)

Ayush Kanodia (#110050049)

Outline

- Aim of project (1 mins)
- Demo (5 mins)
- Teamwork Details (0.5 min)
- Design Details –Algorithm (5 mins)
- Design Details – Implementation (8 mins)
- Viva (9 mins)
- Transition time to next team (2 mins)

Aim of the project

- To compare a conventional algorithm with a meta heuristic algorithm for data compression, in particular image compression
- To implement image compression using neural networks
- To implement image compression using the JPEG compression technique
- To compare results and performance of the two algorithms
- PS: Both algorithms perform lossy compression

Demo

All image compression algorithms fall into three categories

1. Predictive coding
2. Transform coding
3. Vector quantization

The Neural Networks which we have implemented
is an instance of vector quantization

JPEG is an instance of transform coding

JPEG

- As stated, JPEG is an instance of transform coding
- The idea is to convert blocks of pixels (in our case, 8 by 8 blocks) from one basis to another, store a chosen set of coefficients from the new basis, and then reconstruct the image from this chosen basis and chosen coefficients
- JPEG is a fixed basis transform, in that its basis is fixed (cosine functions) for all images

JPEG

With JPEG, it is possible to choose coefficients such that after additional steps, there is drastic compression possible using entropy coding

JPEG

- First, we transform the image from the RGB colour space to the YCbCr colour space. This is because the DCT losses in the YCbCr space are less perceptible to the human eye than those in the RGB colour space. This is a standard matrix transformation.
- After this, the Discrete Cosine Transform is performed on these pixel values
- After performing the discrete cosine transform, a standard matrix is used to quantize the coefficients
- The first coefficient in the block after the DCT is called the DC coefficient. It is the most important coefficient. Successive coefficients are called AC coefficients, and lose importance progressively
- Quantization takes advantage of this, and converts a number of AC coefficients to zeroes

JPEG

- Typical Quantization Matrix

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	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Such matrices are derived purely on the basis of observation, and published by the JPEG (Joint Photographic Experts Group). This is one such matrix

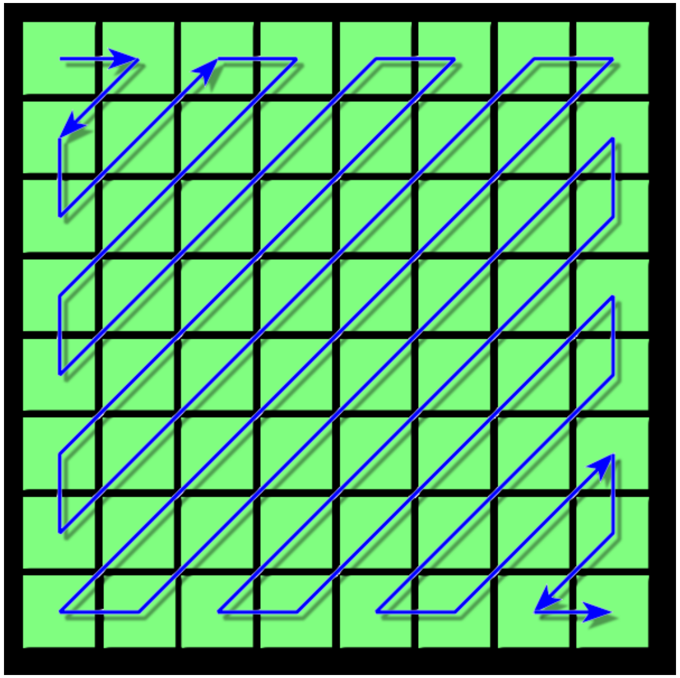
JPEG

- Following this, entropy coding is applied to the quantized data. This part is lossless.
- Entropy coding exploits the fact that different characters from a set of characters (called an alphabet) have different probabilities of occurring
- We first perform RUN LENGTH ENCODING on the quantized data
- This is slightly different from the regular RUN LENGTH ENCODING
- The coordinates are read in a zig zag fashion

JPEG

- Zig Zag Method of traversing blocks

Taken from http://en.wikipedia.org/wiki/File:JPEG_ZigZag.svg



JPEG

- After performing Run Length Encoding, a frequency histogram is created for all possible values obtained
- This is now encoded using Huffman coding
- In our program, we are using a self created Huffman tree, based on the frequency histogram
- After this, the encoded data is stored in a file

JPEG

- Decompression is essentially the inverse process of compression.
- First, the Huffman tree is read from the file.
- Then, using this data, the rest of the file is converted to the RLE data
- After this, the RLE data is converted back to the quantized values
- This marks the end of lossless decompression

JPEG

- Now, the quantized data is used to generate back the DCT Coefficients
- This is followed by the standard inverse Discrete Cosine Transform
- Finally, The pixels are converted back from YCbCr space to RGB space
- These are stored in the target image
- Speciality of our algorithm: Huffman coding is customized

JPEG

- In a nutshell

Credits: The JPEG Still Picture Compression Standard, Gregory K. Wallace, Multimedia Engineering

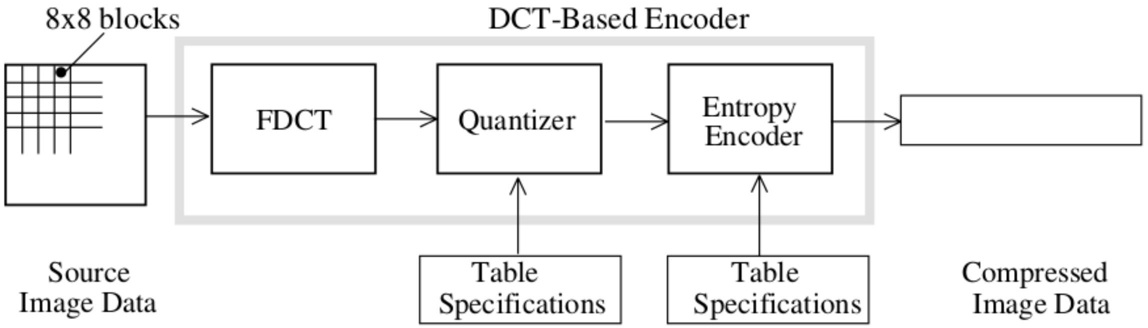


Figure 1. DCT-Based Encoder Processing Steps

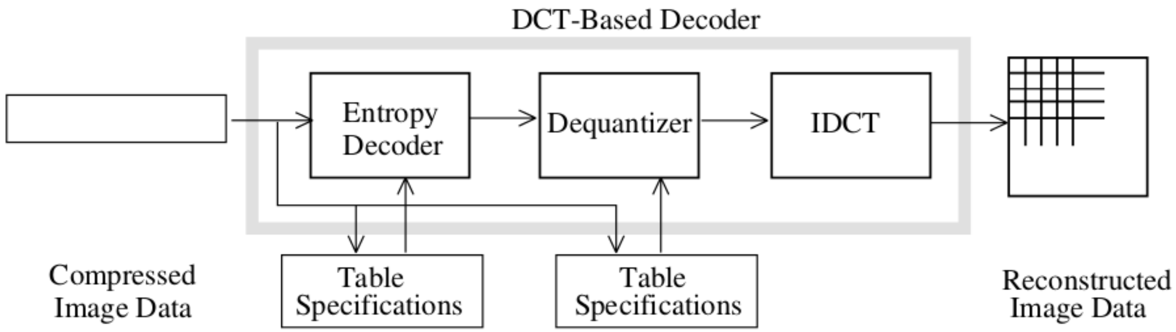


Figure 2. DCT-Based Decoder Processing Steps

JPEG

- Explain if time permits
- Improvements
- Downsampling
- Separate quantization for chroma components
- Another form of run length encoding

Design Details - JPEG

- Algorithm (Note N is the number of pixels)

1. Matrix Transformation to new colour space (O(1) for every pixel, O(N) overall)
2. Discrete Cosine Transform, O(m*m) for every block,

Where m is the number of pixels in the block. For us, m is 8. Since this m is fixed throught out, time taken is O(N) overall.

3. Quantization, O(N) overall
4. Run Length Encoding, O(N) overall
5. Huffman Table creation, O(N) + O(m log m), where is the number of characters in the alphabet we are using. Since m is fixed = 256 for us, this operation is O(N) overall.

(Note: All inverse operations take the same time)

Design Details - JPEG

Implementation

The CImg library used for handling images

Platform – Linux (Ubuntu)

Algorithm Design - JPEG

- Question Statement- Apply JPEG compression to produce the compressed format of an image

The compressed file that we create needs to be directly rendered by an image viewer, but since this algorithm is not standard (there are modifications), and we wished to create our own decompression methods, we also perform decompression

Algorithm Design - JPEG

1. Colour space transformation

Standard matrix transform

AB = C

A is given, 3 * 3, B is the pixel matrix in RGB space, C is the pixel matrix in YCbCr space

Algorithm Design - JPEG

2. Standard Discrete cosine transform and its inverse on every block. The following formula was used. Credits as mentioned

ECE 802 – 602: Information Theory and Coding
Seminar 1 – The Discrete Cosine Transform: Theory and Application

$$C(u,v)=\alpha(u)\alpha(v)\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}f(x,y)\cos\left[\frac{\pi(2x+1)u}{2N}\right]\cos\left[\frac{\pi(2y+1)v}{2N}\right], \tag{4}$$

for $u, v = 0, 1, 2, \ldots, N-1$ and $\alpha(u)$ and $\alpha(v)$ are defined in (3). The inverse transform is defined as

$$f(x,y)=\sum_{u=0}^{N-1}\sum_{v=0}^{N-1}\alpha(u)\alpha(v)C(u,v)\cos\left[\frac{\pi(2x+1)u}{2N}\right]\cos\left[\frac{\pi(2y+1)v}{2N}\right], \tag{5}$$

For detailed explanation, please see,
http://en.wikipedia.org/wiki/JPEG

Algorithm Design - JPEG

3. Quantization and its inverse

Quantization: We have Matrix A and B. We divide every element of A by its corresponding element in B

Inverse Quantization: We have Matrix A and B. We multiply every element of A by its corresponding element in B

Algorithm Design - JPEG

4. Run Length Encoding (RLE) & its inverse

In a zig zag fashion, count the number of zeroes before a non zero number. Then take the first non zero number. The number of zeroes cannot exceed 15 for one RLE unit. Now take the one's complement inverse of the nonzero number, and let the number of bits in this be B. Add B to the above pair to make a triple, and store all such triples in a vector. For its inverse, perform the inverse of each of these steps. Note that B cannot exceed the number 16, by the way the JPEG algorithm is implemented (the quantized values cannot exceed 16)

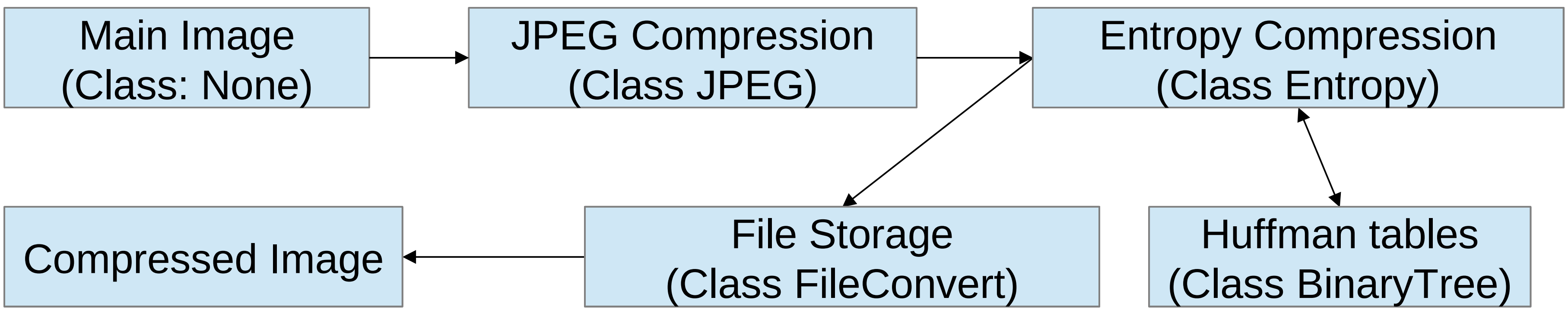
Algorithm Design - JPEG

5. Huffman Coding and its inverse

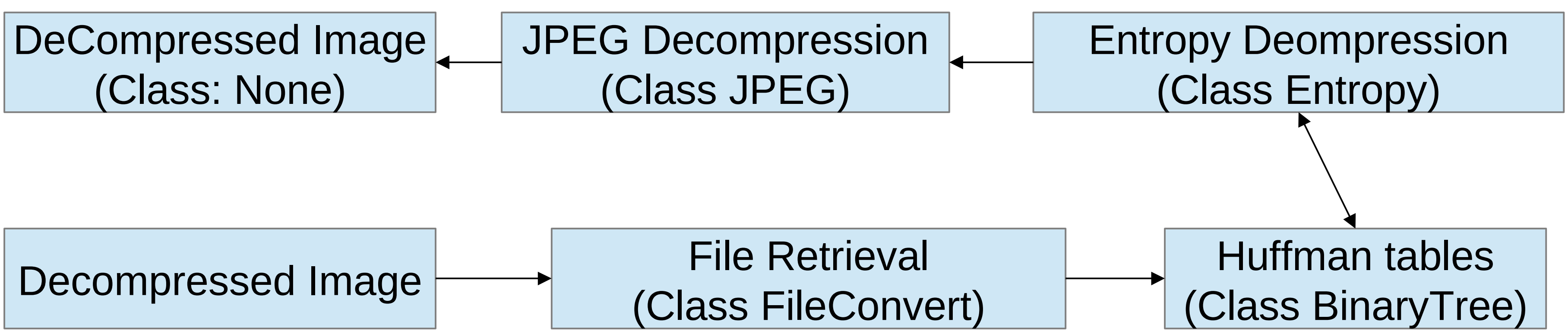
Now, take the pair (the number of zeroes, and the number of bits occupied by the third member of the above triple). There are exactly 256 such pairs (16 * 16). Construct the frequency histogram of this. Using this, create the Huffman table for this data. Now, for every RLE value, store the Huffman key for the first two values of the pair, and then the binary representation of the third member. This will be just a stream of 0's and 1's. Store the Huffman tree and this data, in a file. For the inverse, Read first the tree, and then construct back the run length encoded data using the tree

Class Design (JPEG) – High level

Compression Block Diagram



Decompression Block Diagram



Class Design (JPEG) - Details

Class Name	Brief Description
JPEG	Implements all methods required for JPEG Compression as well as Decompression, including image space transform and its inverse, Discrete Cosine Transform, and its inverse, and Quantization and its inverse
Entropy	Implements all methods for entropy coding. These include Run Length Encoding, and conversion to Huffman Data, and the inverse of these two processes.

Class Design (JPEG) - Details

Class Name	Brief Description
FileConvert	Converts the stream of bits to characters (one character for every eight bits), and its inverse
BinaryTree	Binary Tree operations such as insert, delete, traversal, and leaf checking, for Huffman compression and decompression.

Additional Information – JPEG

- The classes have been designed such that downsampling, and in general, manipulations over a single component can be performed easily, and the code can be easily extended to this. Downsampling has also been tested.
- The Huffman tree is stored in the file very effectively (Explain)
- Instead of using the standard JPEG huffman tree, a new huffman tree, customized to every image, is being used. This gives benefits of improved compression.
- Class design permits easy extension to multiple quantization (explain), different forms of image block scanning(explain), and AC DC coefficient separation(explain).

Data Structures Used (JPEG)

Purpose for which data structure is used	Data Structure Used	Whether Own Implementation or STL
Storing Pixel Values	Array	STL
Storing transform Coefficients	Array	STL
Storing RLE data for a single block of RLE data	Triple	Own
Storing all RLE data	Vector <Triple>	STL
Node of a binary tree	TreeNode	Own

Data Structures Used (JPEG)

Purpose for which data structure is used	Data Structure Used	Whether Own Implementation or STL
Huffman tree	BinaryTree	Own
To make Huffman Tree	Priority Queue	STL

Source Code Information (JPEG)

File Name	Brief Description	Author (Team Member)
Main.cpp	Main coordinating file	Ayush Kanodia
JPEG_Compression.h	Declares JPEG class	Ayush Kanodia
JPEG_Compression.cpp	Implements JPEG class and methods	Ayush Kanodia
Entropy.h	Declares Entropy class	Ayush Kanodia
Entropy.cpp	Implements Entropy class and methods	Ayush Kanodia

Source Code Information (JPEG)

File Name	Brief Description	Author (Team Member)
FileConvert.h	Declares FileConvert Class	Ayush Kanodia
FileConvert.cpp	Implements File Convert Class and methods	Ayush Kanodia
BinaryTree.h	Declares Binary Tree class	Ayush Kanodia
BinaryTree.cpp	Implements Binary Tree class and methods	Ayush Kanodia
Noise.cpp	Implements noise calculation functions	Ayush Kanodia

The slides to follow explain the neural networks design

Design Details

- Algorithm
  - Divide the image into blocks and in each block:-
  - Find some representative set of pixels using which we can represent any pixel(1024 representative pixels in our case)
  - Store the RGB components of only the representative pixels
  - Represent any pixel by the index of the representative pixel, (takes less space to do this)
  - Do huffman encoding and RLE on the pixel values

Design Details

- Implementation
  - Used multi-threading to speed up the process of compression by upto 2-2.5 times. Used the standard thread library of C++ to implement threading in C++ which starts using the power of present-day computers
  - Used the CImg library to read pixel values from image and write a new pixel
  - Used ImageMagick library(used by CImg library) to read JPEG files

Algorithm Design

- Problem : Compressing an image using vector quantization onto a file, and decompressing(rendering the compressed image)
  - Basically the compressed file that we create is supposed to be rendered by an image viewer, (the way JPEG compressed images are directly rendered by the image viewer), but since it is not a standard therefore we also created decompressing algorithms to convert it to a format which the image viewers can read

Algorithm Design

- Divide the image into blocks
  - Divide the image into blocks such that the height and width of each block is approx 200px
  - Formula used :-
    - Num_of_divisions in width = totalWidth/200 + 1
    - Num_of_divisions in height= totalHeight/200 +1
  - Why?
    - We are representing the image using a fixed number of representative pixels, if the block size is too large(or the whole image), then 1024 representative pixels become too less to represent the image colours vividly.

Algorithm Design

- If we use more than 1024 colours, then the algorithm takes more time(linear in number of colours used). Also more bits will be required to store the index of each pixel and hence more space will be required
- However if we use more block, then only we will have to store the representative vector for each block. However this takes very little space and hence doesnt add to overall space complexity.. and hence this choice
- As we will see later the complexity of algorithm is O(number_of_colours_(1024) * number_of_pixels_in_a_block * number_of_blocks)
- This translates to O(num_colors*tot_num_of_pixels)

Algorithm Design

- Find the representative set of pixels. Use of neural networks in this part
  - Neural networks are capable of learning from input information and optimizing itself to correctly give the output as required
  - In our case, 1024 neurons are designed to compute the vector-quantization code-book in which each input vector (pixel in our case) relates to one neuron using the coupling weights.
  - The coupling weight associated with the ith neuron is trained and represented by code-word (which in our case is just the index of the neuron)

Algorithm Design

- As the neural network is being trained, all the coupling weights will be optimized to best represent the possible partition of all the input vectors
- To train the network, the image block is designated as the training set , and any random 1024 input vectors(pixel in our case) are used to initialize all the neurons weights
- Then we iterate through all the input vectors and the competitive learning algorithm is employed by which the output of that neuron whose distance from the input vector is the minimum is 1 and it is called as winning neuron
- The weights of winning neuron is modified as per the following equation :-

w(i) new = w(i) old + alpha* (x – w(i) old)

- The weight of the rest of neuron remains unchanged

Algorithm Design

- The equations mathematically are

  where d(x,W(i)t) is the distance between the input weight vector and the coupling weight vector, alpha is the learning rate and z(i) is the output

$$z_i = \begin{cases} 1 & d(x, W_i(t)) = \min_{1 \le j \le M} d(x, W_j(t)) \\ 0 & otherwise \end{cases}$$

$$W_i(t+1) = W_i(t) + \alpha(x - W_i(t))z_i$$

Algorithm Design

- Assign a code-word to each input vector(pixel)
  - Find the winning neuron for each pixel and assign the index of the winning neuron to represent the pixel
  - Since there are only 1024 neurons, so the index is maximum of 10 bits and hence takes only takes up, in contrast to the 24 bits required to store the pixel (8 bits for each of the red, green and blue components)

Algorithm Design

- Apply huffman coding on the values of each pixel(value=index of winning neuron for that pixel)
  - Why? The neurons are supposed to partition the input vectors such that each input vector gets a representative neuron. It is not necessary that all neuron will be representative of same number of input vectors
  - Hence there is a possibility of using huffman coding to take less space to represent more frequently used neurons and more space to represent less frequently used neurons

Algorithm Design

- Run length encoding
  - Since nearby input vectors are likely to be similar and hence having the same winning neuron, hence the run length encoding is likely to give good results

Multi-threading

- Since operations on one block are completely independent of the other block, hence implementing multi-threading will finish the work in shorter time.
- A normal c++ program is handled by just one thread.
- However , if we implement multi-threading, then the speed of the same c++ program can go upto 4 times. (Most computers these days have multiple cores, and each core can support upto two threads)

Multi-threading

- However multi-threading has various overloads and therefore practically the speed of the same C++ program goes up by about 2- 2.5 times in a standard PC.
- So we have implemented multi-threading in our program , where we have stored the output of each block while processing, and then written it to file at the end. (We cannot write to file simultaneously because then different threads will conflict over the same file)

Algorithm Design

- Decryption of the file
  - Read the coupling weights of all the vectors stored in the file
  - Read the huffman tree stored in the file
  - Read the value stored for each pixel using the huffman tree (and run_length encoding factor to be implemented)
  - Set the pixel of the image to be just the coupling weights of the neuron that it represents

Class Design – High level

- Major functions of compression and decompression are handled by functions
- Huffman_read and huffman_write classes handle all the huffman-and-rle related work. Infile and outfile classes handle all the bit-manipulation type work. Neuron class handles all the work related to neurons. Tree class handles all the manipulations of the tree.

Class Design – High Level

- Main function calls compress or decompress function
- Compress function- uses the neuron class to create new neurons, initialize and modify them , then it gives all the values of pixels to the huffman class.
  - Huffman class – takes in all the values and assigns the huffman codes, and writes it to the outfile object alongwith number_of_bits
  - Outfile class – takes in values and writes to real file after padding the end with zeroes

Class Design – High Level

DeCompress function- Uses the infile object to read parameters like number_of_neurons, maximum_length_of_rle from file. It asks the huffman_read class object to read the formatted huffman_tree. Then it asks the huffman_read class to give values one by one, which it writes to the new image

huffman_read – While reading the formatted tree, it makes an object of tree and while reading the values from file, it uses this tree.

Class Design - Details

Class Name	Details
huffman_read	*Contains the functions to read the huffman_tree from the formatted data, and to read individual values from the file using tree*
huffman_write	Takes in all the values that need to be written to the file, it calculates the distribution of the input and then decides the huffman codes.Once done, we can ask it to write everything to a file in the correct format which can be read back meaningfully

Class Design - Details

Class Name	Details
Outfile	*outfile is a stream which can store the values in a given number of bits, and can write to a real file as reqd*
infile	Infile is a stream which can read a given number of bits from the file
neuron	Stores the coupling weights and some functions of the neurons
Tree	Stores the tree and all related functions

Cool Tricks

- Implemented percentage of completion of the compression because compression takes a lot of time. Percentage of completion of compression appears after constant time, i.e the percentage of completion occurs more frequently if it is a large image and appears less frequently if it is a small image

Cool Tricks

- Comparing the distance square while finding the winning neuron, this is saving upto 5% of time of compression
- A cool way to represent the huffman_tree :- using 0s and 1s to represent the edges (0 if the edge end is not a leaf and 1 if image end is a leaf.Storing the value in 10 bits when a leaf is encountered)

Data Structures Used

Purpose for which data structure is used	Data Structure Used	Whether Own Implementation or STL
To make the huffman tree	*Priority queue*	*STL*
To represent huffman_tree	Tree	*Own*
To read characters from the file based on the number_of_bits_reqd	Stack	STL
To represent the huffman code of any value	List	STL
To represent the characters to be written to the file	String	STL

Source Code Information

File Name	Brief Description	Author (Team Member)
Encrypt.cpp	*Contains the compression functions*	*Navin Chandak*
Decrypt.cpp	*Contains the function for decompression*	*Navin Chandak*
Index.cpp	Contains the main function which redirects to encyrption or decompression	Navin Chandak
Neuron.h	Declares the neuron class, which contains all the data of a particular neuron	Navin Chandak
Neuron.cpp	Defines the neuron functions	Navin Chandak

Source Code Information

File Name	Brief Description	Author (Team Member)
File.h	*Declares the infile and outfile class*	*Navin Chandak*
File.cpp	*Defines the methods of infile and outfile class*	*Navin Chandak*
Huffman.h	Declares the huffman_read and huffman_write class	Navin Chandak
Huffman.cpp	Defines the methods of huffman_read and huffman_write	Navin Chandak
Tree.h	Declares the tree class	Navin Chandak
Tree.cpp	Defines the methods of tree	Navin Chandak
Vectorf.h	Declares functions of vectors	Navin Chandak
Vectorf.cpp	Defines vector functions	Navin Chandak

Analysis

Here is a comparative time and performance analysis of the two algorithms. We have performed time and performance analysis on lena, the standard test image used for image compression algorithms, and 24 images released by Kodak for this purpose. These are named as com<i>.png, “i” running from 1 to 24. The best and worst cases are displayed here

Analysis

The measure of goodness of an algorithm is the peak signal to noise ratio (PSNR). Apart from this, the running time has also been measured and documented

It is defined as PSNR = 20 log (MAX / (sqrt. MSE), for each colour component

MAX = 255, the maximum colour component

MSE is defined as $$MSE = \frac{1}{m\,n}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[I(i,j) - K(i,j)]^2$$

m is the Height, n is the width, I standards for original image, K for reconstructed Image. Credits: www.wikipedia.org

lena.bmp

Neural Network Compression Time
real 0m13.704s
Neural Network Decompression Time
real 0m2.159s
Neural Network Performance
PSNR RED = 41.3265 PSNR GREEN = 41.2804 PSNR BLUE = 41.1619
JPEG Compression Time
real 0m4.584s
JPEG Decompression Time
real 0m4.557s
JPEG Performance
PSNR RED = 34.634 PSNR GREEN = 33.9264 PSNR BLUE = 31.8346
File Sizes in kilobytes
Original File Size
786
Neural Network Size
361
JPEG Size
59

Best Performance

Neural Networks

Note: This image has very few colours

Image com16.png
Neural Network Compression Time
real 0m24.468s
Neural Network Decompression Time
real 0m2.897s
Neural Network Performance
PSNR RED = 47.8072 PSNR GREEN = 47.7347 PSNR BLUE = 47.4245
JPEG Compression Time
real 0m6.666s
JPEG Decompression Time
real 0m6.650s
JPEG Performance
PSNR RED = 33.6353 PSNR GREEN = 33.9281 PSNR BLUE = 33.4116
File Sizes In kilobytes
Original File Size
1152
Neural Network Size
493
JPEG Size
67

Best Performance

JPEG Compression

Note: This image has similar colours arranged in blocks

Image com23.png
Neural Network Compression Time
real 0m24.481s
Neural Network Decompression Time
real 0m2.460s
Neural Network Performance
PSNR RED = 40.517 PSNR GREEN = 40.8961PSNR BLUE = 40.4858
JPEG Compression Time
real 0m6.699s
JPEG Decompression Time
real 0m6.633s
JPEG Performance
PSNR RED = 36.5026 PSNR GREEN = 37.0811 PSNR BLUE = 36.0093
File Sizes in kilobytes
Original File Size
1152
Neural Network Size
415
JPEG Size
67

Worst Performance

Neural Networks

Note: This image has numerous colours

Image com5.png
Neural Network Compression Time
real 0m22.772s
Neural Network Decompression Time
real 0m2.976s
Neural Network Performance
PSNR RED = 39.8704 PSNR GREEN = 39.9062 PSNR BLUE = 39.8029
JPEG Compression Time
real 0m6.724s
JPEG Decompression Time
real 0m6.688s
JPEG Performance
PSNR RED = 30.287PSNR GREEN = 30.5597 PSNR BLUE = 30.0898
File Sizes in kilobytes
Original File Size
1152
Neural Network Size
498
JPEG Size
114

Worst Performance

JPEG Compression

Note: This image has many variations close together, in the same block

Image com13.png
Neural Network Compression Time
real 0m23.999s
Neural Network Decompression Time
real 0m3.326s
Neural Network Performance
PSNR RED = 42.9397 PSNR GREEN = 43.2172 PSNR BLUE = 42.7656
JPEG Compression Time
real 0m6.718s
JPEG Decompression Time
real 0m6.677s
JPEG Performance
PSNR RED = 27.9091 PSNR GREEN = 28.0006PSNR BLUE = 27.601
File Sizes in kilobytes
Original File Size
1152
Neural Network Size
550
JPEG Size
115

Complete Analysis

The complete analysis is present in a file called analysis.pdf . Please see it for details.

Brief Conclusion

We have implemented and analysed the two methods for image compression. It is clear that

1. JPEG compression is better for compressing images that do not require detailing, have not many sharp edges and need high compression (Typical example : Images that need to be transmitted over the web). Also, it takes less compression time, but decompression time is equivalent for compression
2. The Neural Network Method is better for images with better detailing, and substantial image compression is achieved. Decompression and rendering is very fast, although compression time is very high.

(Typical example of usage: Archiving pictures which need to be viewed numerous times)

Teamwork Details

- Navin Chandak (#110050047)

Complete implementation of Meta heuristic algorithm (Using Neural Networks)

- Ayush Kanodia (#110050049)

Complete implementation of Conventional Algorithm (Using JPEG)

- Comparison, analysis and documentation done together

Overall Contribution by Navin Chandak	Overall Contribution by Ayush Kanodia
50 +- 5 %	50 +- 5%

Thank You – Questions?

Back Up Slides