CS663 - Project: JPEG Image compression

Keshava Kotagiri, Kuchipudi Rahul Deepak, Rajkumar

210050088, 210050090, 210050129

November 2024

1 Introduction:

This project focuses on compressing images, both color and grayscale, using the JPEG algorithm. The process begins by reading an input image and applying the JPEG algorithm, which involves zeroing out high-frequency components to retain essential image details while removing less noticeable information. The compressed image data is encoded into a binary string and stored in a file. A decompression function reads this binary file to reconstruct the image. The reconstructed image is lossy compared to the original due to the nature of the JPEG algorithm.

Furthermore,we have analyzed the compression efficiency by plotting the Root Mean Squared Error (RMSE), which measures the loss between the original and reconstructed images, against Bits Per Pixel (BPP), and tells how much the image has been compressed. We have plotted like this for 20 images.

2 Implementation:

JPEG Compression Process

1. **Input Image Processing:** Start with the input image (grayscale or color). For color images, convert them to the YCbCr format and downsample the Cb and Cr components if needed.
2. **Block Division:** Divide the image into 8×8 blocks. Add padding to ensure dimensions are multiples of 8 if necessary.
3. **Discrete Cosine Transform (DCT):** Apply the DCT to each 8×8 block to transform spatial data into frequency components.
4. **Quantization:** Divide the DCT coefficients of each block by a quantization matrix determined by a chosen quality factor. Round the results to reduce high-frequency details and data size.
5. **Encoding:** Arrange the quantized values in a zig-zag order, which groups zeros together. Apply Huffman encoding to compress the data efficiently.
6. **Storage:** Save the compressed data, along with necessary metadata such as quantization and Huffman tables, into a binary file.
7. **Decompression and Reconstruction:** Read the binary file, decode the data, reverse quantization by multiplying with the quantization matrix, and apply the Inverse DCT (IDCT) to reconstruct the blocks. Finally, reassemble the blocks to recreate the image.

3 Code files:

We have 5 code files we are using. Each code file's working is explained below:

3.1 compress.py

The `compress()` function is designed to take an image filename and a quantization matrix as input. It processes the image and saves a `.bin` file containing the compressed data. The function does not directly return any value but outputs a file that represents the compressed image.

First, the function checks if the input image is grayscale or color. While both types are handled similarly, we will describe the grayscale compression process here, the color image process includes a few additional steps explained in later section.

The image is padded to ensure its dimensions are multiples of 8. This padding is necessary because the compression process involves dividing the image into 8×8 blocks. Once padded, the image is split into these non-overlapping blocks.

Next, a **2D-DCT (Discrete Cosine Transform)** is applied to each block. The resulting coefficients are divided by a **quantization matrix** with a quality factor of choice. The values are then rounded to integers to remove high-frequency components, which contribute less to the perceived image quality.

Each block is then read in a **zig-zag order**, producing a 64-element array. This step often results in many trailing zeros. These zeros are replaced by an **EOB (End of Block)** marker with the value 32767 (or $2^{15} - 1$). All blocks are concatenated into a single list, separated by EOB markers.

The concatenated data is further compressed using **Huffman encoding**. This step converts the list of numbers into a compact **bit-string**, including a Huffman table for decoding.

Finally, metadata such as the image dimensions and a flag indicating whether the image is grayscale (0) or color (1) is prepended. The resulting bit-string is saved to a `.bin` file using the **binary.py** module, ready for decompression.

3.2 huffman.py

The Huffman module contains two key functions: `huffman_compress()` and `huffman_decompress()`. The former takes a list or array of numbers and produces a bit string, while the latter reverses the process, converting a bit string back into the original list of numbers. These functions are integral to the workflows of `compress.py` and `decompress.py`, respectively.

The **Huffman algorithm** efficiently encodes a list of numbers based on their frequency, minimizing storage requirements. However, since the encoded data must be saved as a binary file, the Huffman table itself also needs encoding as part of the bit string.

To achieve this, a **20-bit header** is prefixed to the bit string. This header specifies the number of bits used to encode the Huffman table, ensuring compatibility during decompression. Notably, the size of the encoded table cannot exceed 2^{20} bits.

The returned bit string, composed entirely of 0s and 1s, contains both the encoded Huffman table and the compressed representation of the input numbers.

The `huffman_decompress()` function begins by reading the first 20 bits to extract the table's size. It then decodes the Huffman table and subsequently reconstructs the original list of numbers, returning the decompressed array.

3.3 binary.py

This has two main functions, save_to_file() and load_from_file().

As we said before, a compressed version of the image is saved in a binary file. One of the problems that

arises is that a file in a computer is saved in bytes and a .bin file saves raw data. But our data is in bits and isn't necessarily in multiples of 8. Hence we decided to put three bits at the start which tell how much padding is required and then put padding at the end to make it multiple of 8.
When file is loaded, it checks the first three bits, and removes the padding, then returns the original bit string.

3.4 decompress.py

This basically does all the steps as before, but in reverse order - get back binary string from .bin file, check if color or grayscale using first bit, check image dimensions, then separate blocks, replace EOB with trailing zeros, reverse the zig-zag to get back 8×8, multiply by quantization matrix, perform IDCT, then rejoin blocks (as per image dimensions found at start) to get back reconstructed image.

3.5 user.py

This is the code which we are using for running and testing our JPEG compression algorithm, plotting RMSE vs BPP plots etc.

4 Color Implementation

A color image is basically three grayscale images - R, G, B. We have implemented compressing directly from RGB and well as converting it into YCbCr and then downsampling Cb and Cr and then compressing it. **user.py** asks if it is a color image on how which type to proceed into.
In RGB case, each of the three is compressed similar to a grayscale and then recombined.
In case of YCbCr, we have downsampled for Cb and Cr such that for every 4 horizontal pixels only 2 are taken, while for every 2 vertical pixels, 1 is taken. Then each of Y, Cb, and Cr are compressed separately (though their compressed bit strings are still concatenated in the .bin file). At the end we have to resample Cb and Cr because or else their sizes are half of Y due to previous downsampling.

5 Results

We tested our code for color and grayscale on 20 images each. with different quality factor values {10, 30, 50, 70, 90}

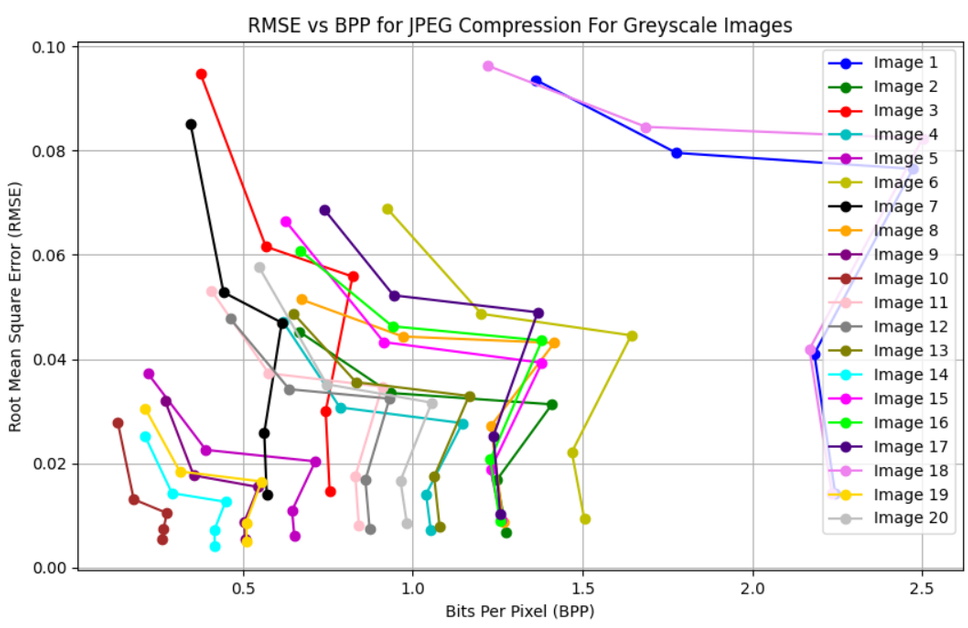

Figure 1: RMSE vs BPP line plots for grayscale images.

Our submission file directory structure is:

```
Submission Folder/
  |--color_images_dataset
  |--gray_images_dataset
  |-binary.py
  |-compress.py
  |-decompress.py
  |-huffman.py
  |-binary.py
  |-user.py
  |-generate_graph.py
  |-run_compressor.sh
  |-run_graph.sh
  |-kodak24.jpg
  |-kodim24.png
  |-report.pdf
```