Image Compression Engine: Implementation and Analysis

Course Project Report - CS663

Harsh | Pranav | Swayam

November 25, 2024

Contents

1 Introduction **2**

2 Problem Statement **2**

3 Algorithm Description **2**

4 Dataset Description **3**

5 Results and Analysis **3**

5.1 Discussion 3

6 Conclusion **4**

7 Introduction **4**

8 Code Implementation and Explanation **4**

8.1 Code for JPEG-Like Compression 4

8.2 Explanation of Code Sections 4

9 Results **6**

10 Discussion **7**

10.1 Strengths 7

10.2 Limitations 7

11 Conclusion **7**

Abstract

This report describes the implementation of an image compression engine inspired by the JPEG algorithm. The implemented system performs discrete cosine transform (DCT) on image patches, quantizes the coefficients, applies Huffman coding, and writes the compressed data to a file. A decoding module reconstructs the image for analysis. Results for multiple images are analyzed using metrics like RMSE and BPP across varying quality factors. The project's merits and limitations are also discussed.

1 Introduction

Image compression is a critical area in digital image processing, where the objective is to reduce the size of image files without significant loss of quality. The JPEG compression standard achieves this using techniques such as Discrete Cosine Transform (DCT), quantization, and entropy coding. This project implements a simplified version of the JPEG compression algorithm, evaluates its performance, and analyzes its strengths and weaknesses.

2 Problem Statement

The objective of this project is to:

- Implement a simplified JPEG-like compression algorithm for grayscale images.
- Evaluate the algorithm's performance by calculating metrics such as Root Mean Square Error (RMSE) and Bits Per Pixel (BPP).
- Generate reconstructed images for various quality factors and analyze the trade-off between compression ratio and image quality.

3 Algorithm Description

The implemented algorithm consists of the following steps:

1. **Discrete Cosine Transform (DCT):** The input image is divided into non-overlapping 8×8 blocks, and the 2D DCT is applied to each block to transform the image into the frequency domain.
2. **Quantization:** A quantization table is used to reduce the precision of the DCT coefficients. The quantization table values are scaled based on a user-defined quality factor.
3. **Run-Length Encoding (RLE):** The quantized DCT coefficients are scanned in a zigzag order, and sequences of zero coefficients are encoded efficiently using run-length encoding.

4. **Huffman Encoding:** The RLE output is further compressed using Huffman encoding, which replaces frequently occurring patterns with shorter codes.
5. **Decoding:** The compressed data is decoded by reversing the Huffman and RLE encoding processes, followed by inverse quantization and the Inverse Discrete Cosine Transform (IDCT).
6. **Reconstruction:** The reconstructed image is generated and compared with the original image to calculate the RMSE and evaluate the compression ratio.

4 Dataset Description

The algorithm was tested on a dataset of grayscale BMP and TIF images, converted from color images if necessary. Each image had varying dimensions and complexity to evaluate the algorithm's robustness. The dataset included:

- SegPC-2021: Segmentation of Multiple Myeloma Plasma Cells in Microscopic Images.
- "Standard" test images (a set of images found frequently in the literature) from Image Processing Place.

The images were processed for five quality factors: 10, 20, 50, 75, and 90.

5 Results and Analysis

5.1 Discussion

The algorithm performs well in reducing file size while maintaining acceptable image quality. However:

- **Strengths:**
  - Efficient use of frequency domain for compression.
  - Effective reduction in redundancy using RLE and Huffman encoding.
  - Customizable quality factor allows flexible trade-offs.
- **Weaknesses:**
  - High computational overhead due to block-wise DCT and Huffman encoding.
  - Loss of detail in high-frequency regions at low quality factors.
  - Limited optimization for real-time applications.

6 Conclusion

The implemented JPEG-like compression algorithm successfully demonstrates the principles of lossy image compression. The RMSE and BPP results confirm the effectiveness of the approach in achieving a balance between image quality and compression ratio. Future work could involve:

- Extending the implementation to color images.
- Incorporating advanced entropy coding techniques like arithmetic coding.
- Optimizing the algorithm for real-time processing.

7 Introduction

Image compression is a critical aspect of digital storage and transmission. The JPEG algorithm achieves significant compression by exploiting redundancies in image data while maintaining perceptual quality. This project implements the essential components of JPEG compression on grayscale images and evaluates its performance using RMSE and BPP metrics.

8 Code Implementation and Explanation

This section explains the MATLAB code for JPEG-like image compression. The steps include loading an image, applying DCT, quantization, Huffman encoding, and reconstruction. RMSE and BPP are calculated to evaluate compression performance.

8.1 Code for JPEG-Like Compression

```matlab
% Set image directory path
image_dir = '../images/';
...
% Create a new figure for the second plot
figure;
plot(bpp_values, rmse_values, '-o');
xlabel('Bits Per Pixel (BPP)');
ylabel('RMSE');
title('RMSE vs. BPP');
```

Listing 1: Image Compression Code

8.2 Explanation of Code Sections

1. Reading the Image: The image is loaded and converted to a double format for accurate DCT calculations. The block size for partitioning the image into non-overlapping patches is set to 8x8.

```
image = imread(fullfile(image_dir, 'kodak24.png'));
image = double(image);  % Convert to double for DCT
blockSize = [8 8];
```

Listing 2: Reading the Image

2. Applying DCT: The blockproc function applies 2D DCT to each 8x8 block of the image. This step transforms spatial information into frequency components.

```
dct_image = blockproc(image, blockSize, @(block) dct2(block.data));
```

Listing 3: Applying 2D DCT

3. Quantization: Quantization reduces the precision of the DCT coefficients using a predefined quantization matrix. A higher scaling factor reduces quality but increases compression.

```
quantization_matrix = [
    16, 11, 10, 16, 24, 40, 51, 61;
    12, 12, 14, 19, 26, 58, 60, 55;
    ... % truncated for brevity
];
quantized_image = blockproc(dct_image, blockSize, @(block) round(
    block.data ./ quantization_matrix));
```

Listing 4: Quantization Step

4. Huffman Encoding: The quantized coefficients are flattened and encoded using Huffman coding to minimize storage. The encoded data and metadata are saved to a file.

```
symbols = unique(quantized_image(:));
counts = histcounts(quantized_image(:), [symbols; max(symbols)+1]);
[dict, avglen] = huffmandict(symbols, counts / numel(
    quantized_image));
huff_encoded_image = huffmanenco(quantized_image(:), dict);
save('compressed_image.mat', 'huff_encoded_image', '
    quantization_matrix', 'dict', 'size_of_image', '-v7.3');
```

Listing 5: Huffman Encoding and Saving

5. Reconstruction: The compressed data is decoded, dequantized, and the inverse DCT is applied to reconstruct the image.

```
decoded_data = huffmandeco(huff_encoded_image, dict);
quantized_image = reshape(decoded_data, size_of_image);
dequantized_image = blockproc(quantized_image, blockSize, @(block)
    block.data .* quantization_matrix);
```

```
reconstructed_image = blockproc(dequantized_image, blockSize, @(
    block) idct2(block.data));
```

Listing 6: Reconstruction

6. Calculating Metrics: RMSE and BPP are calculated to analyze compression quality. RMSE measures error between original and reconstructed images, while BPP indicates compression efficiency.

```
rmse = sqrt(mean((double(image) - double(reconstructed_image)).^2,
    'all'));
bpp = numel(huff_encoded_image) / numel(image);  % Bits per pixel
```

Listing 7: Metrics Calculation

7. Quality Factor Analysis: A loop adjusts the quantization matrix for different quality factors, recalculating RMSE and BPP for each iteration. A plot of RMSE vs. BPP summarizes the trade-offs.

```
quality_factors = 10:10:100;
for idx = 1:length(quality_factors)
    % Adjust quantization matrix based on quality factor
    scaled_quantization_matrix = round(quantization_matrix * (100 /
         quality_factors(idx)));
    ...
end
```

Listing 8: Quality Factor Analysis

9 Results

10 Discussion

10.1 Strengths

- Efficient implementation of JPEG components.
- Accurate reconstruction of images at moderate quality factors.

10.2 Limitations

- Compression artifacts visible at low quality factors.
- Fixed quantization matrix limits adaptability.

11 Conclusion

The MATLAB implementation effectively demonstrates JPEG-like compression. By varying quality factors, the trade-off between compression efficiency (BPP) and reconstruction quality (RMSE) was analyzed. Future improvements can include adaptive quantization and optimizing encoding schemes.