CS 663 Course Project

Implementation of JPEG image compression algorithm
and Paper Implementation.

Report

by

Saral Sureka

23M2113

Aryan Gupta

22B2255

Under the guidance of

Prof. Ajit Rajwade

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai 400076 (India)

November 24, 2024

Acknowledgements

We would like to express our sincere gratitude to our course instructor, Prof. Ajit Rajwade, for his invaluable guidance, expertise, and continuous support. His insights and feedback greatly contributed to the development of this work.

Additionally, we would like to acknowledge the use of various libraries, tools, and resources that were essential for the implementation of the algorithms and experiments. Without them, the successful completion of this project would not have been possible.

Abstract

This report explores the implementation and analysis of two image compression techniques: JPEG compression and edge-based image compression using homogeneous diffusion. The first part of the report delves into the fundamental components of JPEG compression, including the 2D Discrete Cosine Transform (DCT), quantization, zig-zag traversing, Run-Length Encoding (RLE), and Huffman encoding. It provides a detailed breakdown of the project, simulation methodologies, and evaluation metrics, culminating in a discussion of compression efficiency and quality.

The second section is implementation of a paper which suggests a novel edge-based compression algorithm that utilizes homogeneous diffusion for encoding and decoding image data. This approach focuses on detecting edges, encoding contour locations and pixel values, and reconstructing missing data during decompression. The experimental analysis evaluates the performance of both compression techniques with respect to image quality and compression ratio. Results highlight the trade-offs between compression efficiency and image fidelity, and visual comparisons underscore the effectiveness of each method under varying conditions. The report concludes with insights on the applicability of these techniques for different types of image data.

Contents

Chapter 1

Introduction to JPEG Compression

JPEG (Joint Photographic Experts Group) [7] is a widely used method of lossy image compression standard, primarily designed for compressing photographic images. It was developed in the early 1990s by the JPEG committee and has become the most common format for storing and transmitting digital images, particularly in web applications and digital photography.

The primary goal of JPEG compression is to reduce the file size of images without significantly degrading their perceived quality. This compression technique exploits the limitations of human visual perception by removing information that is less noticeable to the human eye. It achieves high compression ratios by applying several stages of image processing, including transformation, quantization, and encoding.

JPEG compression can be described as a *lossy* method, meaning that some of the original image data is discarded during compression. This results in a reduction in file size, but also a loss of detail in the image. However, the loss is often imperceptible to the human observer, particularly when the compression is applied at moderate levels.

The JPEG algorithm is typically implemented in a series of steps:

1. **Color Space Conversion**: The image is often converted from RGB (Red, Green, Blue) color space to YCbCr (Luminance, Chrominance), as the human eye is more sensitive to luminance than chrominance.

2. **Blocking**: The image is divided into small 8x8 blocks of pixels. Each block is compressed independently to improve computational efficiency.

3. **Discrete Cosine Transform (DCT)**: Each 8x8 block is transformed using the DCT, which converts the spatial domain information into frequency domain coefficients.

4. **Quantization**: The DCT coefficients are quantized, meaning that less important frequencies are rounded or discarded, resulting in data loss.

5. **Entropy Encoding**: The quantized values are further compressed using entropy encoding algorithms like Huffman coding or Arithmetic coding.

JPEG allows for adjustable compression settings, giving users control over the trade-off between image quality and file size. At higher compression ratios, the image may show noticeable artifacts such as blurring, blockiness, or color distortions, while at lower compression ratios, the quality remains high, but the file size is larger.

Despite its lossy nature, JPEG compression is highly efficient and remains the standard for compressing photographic and realistic images, offering a good balance between quality and file size.

Chapter 2

Implementation of JPEG Compression

This chapter discusses the implementation of key steps involved in JPEG compression and the evaluation of its performance. The primary metrics used for evaluating the performance are Root Mean Squared Error (RMSE) and Bits Per Pixel (BPP). The implementation follows the standard JPEG compression process, and includes the following steps:

2.1 Project Breakdown

The implementation is divided into several key stages, each of which contributes to the overall JPEG compression process. These stages are detailed below.

2.1.1 2D Discrete Cosine Transform (DCT)

The Discrete Cosine Transform (DCT) [1] is a fundamental operation in JPEG compression, used to transform spatial domain data into frequency domain coefficients. In JPEG, this transformation is applied to 8x8 blocks of image pixels. The following steps are involved in the DCT implementation:

- Use an existing library like `scipy.fftpack.dct` or manually implement the 2D DCT and its inverse (IDCT) [6].
- Divide the image into non-overlapping 8x8 blocks, a standard size used in JPEG compression.
- Apply the 2D DCT to each block independently to transform it into frequency components.

The DCT is critical for concentrating most of the image's information into a small number of coefficients, allowing for efficient compression.

2.1.2 Quantization

Quantization is the process of reducing the precision of the DCT coefficients, which leads to the lossy nature of JPEG compression. The quantization process is done using a predefined quantization table or by scaling it based on a quality factor. The key steps involved in quantization are:

- Use standard JPEG quantization tables, or allow for adjustable quality factors to scale the table. The quality factor typically ranges from 0 (low quality, high compression) to 100 (high quality, low compression).

- Quantize the DCT coefficients by dividing each coefficient by the corresponding value in the quantization table and rounding to the nearest integer.
- For decompression, the coefficients are *dequantized* by multiplying the quantized values with the same quantization table.

2.1.3 Zig-Zag Traversing

To prepare the quantized coefficients for encoding, JPEG uses a zig-zag [4] pattern to traverse the 8x8 block. This zig-zag scan is a method of flattening the 2D matrix of quantized DCT coefficients into a 1D array. The zig-zag scan prioritizes low-frequency coefficients (which are typically more important) and places them at the beginning of the array. This facilitates better compression in subsequent encoding stages.

To understand the zig-zag traversal, let's consider a simple 3x3 matrix as an example:

$$\text{Matrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Zig-zag scanning involves traversing the matrix in a zig-zag pattern, starting from the top-left corner and moving diagonally downwards and upwards. The zig-zag order for this matrix is as follows:

$$[1, 4, 2, 3, 5, 7, 8, 6, 9]$$

2.1.4 Run-Length Encoding (RLE)

Run-Length Encoding (RLE) is used to compress the sequence of quantized and zig-zagged coefficients. After the zig-zag traversal, many coefficients are often zeros (especially the higher frequency components), and RLE efficiently encodes sequences of zeros as a single symbol (i.e., the number of consecutive zeros). This reduces the number of bits needed to represent the image data. The key steps for RLE are:

- Traverse the zig-zagged list of quantized DCT coefficients.
- For sequences of consecutive zeros, encode the number of zeros as a single pair: the number of zeros followed by the next non-zero coefficient.

RLE is particularly effective for JPEG images, as it handles the sparse nature of high-frequency DCT coefficients well.

Consider the following sequence of numbers:

```
[24, 2, 0, 1, 0, 0, -1, 0, -1, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Run-Length Encoding (RLE) compresses this sequence by representing consecutive repeated numbers as pairs of the form $(\text{value}, \text{count})$, where value is the number and count is the number of consecutive occurrences of that number.

The RLE conversion of the sequence results in the following pairs:

$$[(24, 1), (2, 1), (0, 1), (1, 1), (0, 2), (-1, 1), (0, 1), (-1, 1), (0, 1), (-1, 1), (0, 53)]$$

2.1.5 Huffman Encoding

Huffman Encoding [3] is a lossless entropy encoding algorithm used in JPEG to encode the quantized and run-length encoded data. The process involves the following steps:

- Calculate the frequencies of the quantized coefficients, especially the symbols representing runs of zeros and non-zero values.
- Build a Huffman tree based on these frequencies, where more frequent symbols are assigned shorter codes.
- Encode the coefficients using the generated Huffman codes.

Huffman encoding reduces the size of the compressed data by assigning variable-length codes to symbols, with shorter codes assigned to more frequent symbols.

Referring from class notes:

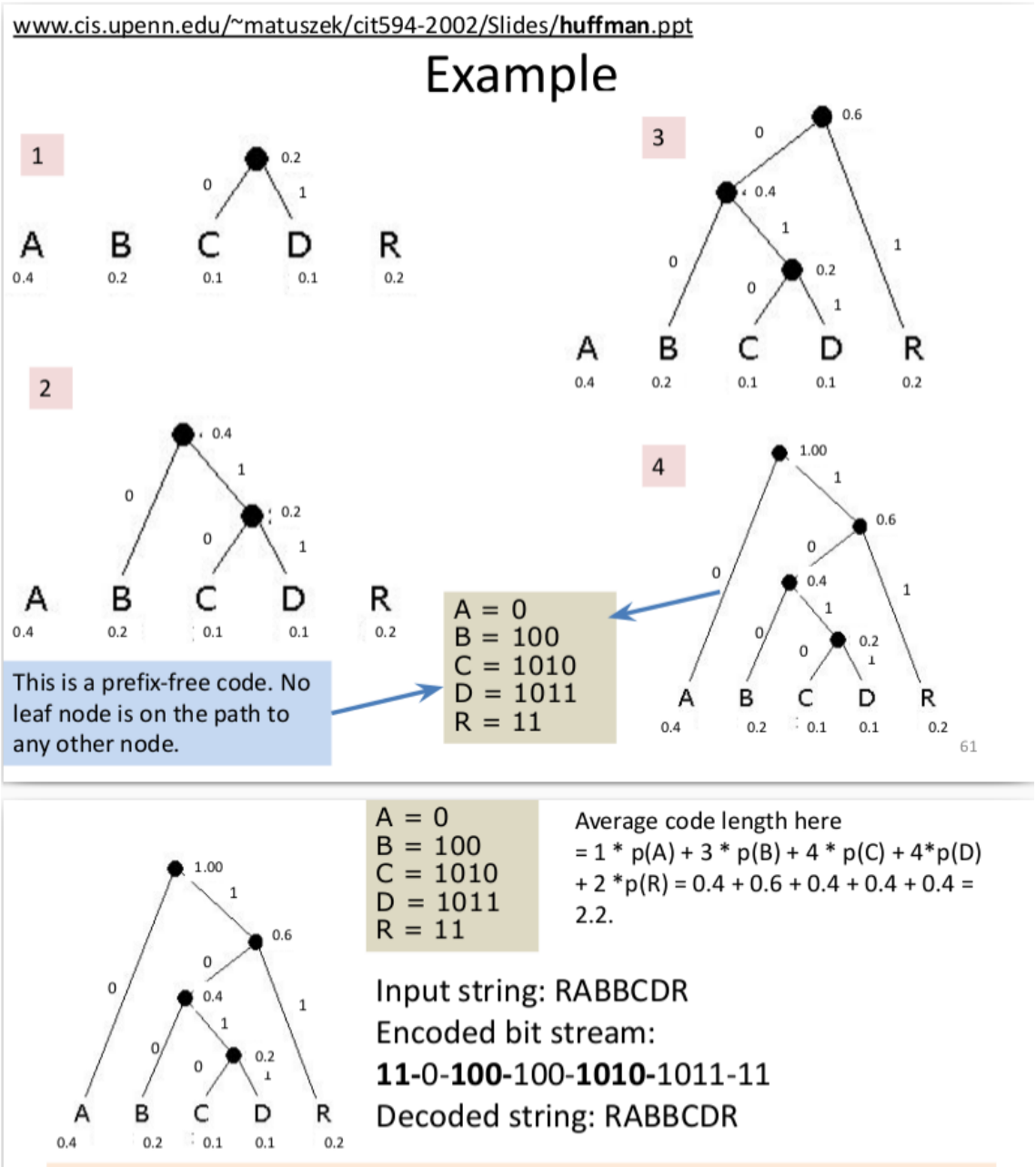


Figure 2.1: An example of Huffman encoding.

2.1.6 File Format

In this step, we define a custom binary file format to store the compressed data. The file format includes essential information for decompression, such as:

- Dimensions of the image (height and width).
- The quantization table used for compression.
- The encoded DCT coefficients, which have been compressed using Run-Length Encoding (RLE) and Huffman coding.

The decompression process involves reading this file, extracting the quantization table and encoded coefficients, and then reconstructing the image.

2.1.7 Simulation and Metrics

To evaluate the performance of the JPEG compression implementation, we test the system with different quality factors. The key performance metrics are:

- **Root Mean Squared Error (RMSE)**: This metric is used to measure the difference between the original image and the decompressed image. It is defined as:

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \hat{x}_i)^2}$$

  where x_i and $\hat{x}_i$ are the pixel values of the original and decompressed image, respectively, and N is the total number of pixels.

- **Bits Per Pixel (BPP)**: This metric is used to quantify the compression efficiency. It is defined as:

$$\text{BPP} = \frac{\text{Total size of the compressed image in bits}}{\text{Number of pixels}}$$

The relationship between RMSE and BPP is evaluated by plotting RMSE versus BPP for different compression levels, which allows us to visualize the trade-off between image quality and file size.

2.2 Conclusion

In this chapter, we discussed the implementation of the JPEG compression algorithm. Key steps including 2D DCT, quantization, zig-zag scanning, run-length encoding, Huffman encoding, and the design of a custom file format were detailed. The compression performance was evaluated based on RMSE and BPP metrics, providing insights into the trade-offs between compression quality and file size.

Chapter 3

Extensive Experiment Analysis on JPEG Algorithm

This report presents the results of an experiment conducted to analyze the importance of each step in the JPEG compression algorithm(on grayscale images). The experiment was designed to compare two methods of flattening the quantized Discrete Cosine Transform (DCT) matrix: one using Zig-Zag traversal followed by Run-Length Encoding (RLE), and the other using a direct row-wise flattening approach. We also studied the impact of varying the Quality Factor (QF) on the resulting compressed images.

The experiment aimed to explore the role of matrix traversal techniques, quantization, encoding, and decoding processes in the overall JPEG compression scheme. The images from the Kodak dataset were chosen for their high quality, and we evaluated how different quality factors affect the visual quality and compression efficiency.

3.1 Dataset

The dataset used in this experiment was the **Kodak Dataset** from Kaggle, which contains grayscale images with high resolution. All images used in the experiment had dimensions of 512x768 or 768x512 pixels, and were selected because their dimensions were divisible by 8 (which is required for the JPEG compression block size).

You can access the dataset here: `https://www.kaggle.com/datasets/sherylmehta/kodak-dataset`.

3.2 Experiment Objective

The main objective of the experiment was to understand how different steps of the JPEG compression process affect image quality and compression efficiency. Specifically, we wanted to compare two approaches for flattening the quantized DCT matrix:

- **Zig-Zag Traversal and RLE (Run-Length Encoding):** This method first flattens the quantized DCT matrix using Zig-Zag traversal and then applies RLE for further compression.
- **Direct Row-Wise Flattening:** This method directly flattens the quantized DCT matrix row by row without any additional encoding.

Additionally, the impact of varying the Quality Factor (QF) from 5 to 90 was analyzed, with the goal of observing how this parameter affects both the image quality and compression ratio.

3.3 Libraries and Tools Used

The following Python libraries and tools were used for image processing, compression, and analysis:

- **numpy:** Used for matrix operations and numerical calculations.
- **scipy:** Provided the implementation of Discrete Cosine Transform (DCT) and Inverse DCT (IDCT).
- **Pillow:** Used for image loading and saving.
- **matplotlib:** Utilized for plotting graphs to analyze results.
- **huffman:** Used for Huffman encoding and decoding.

3.4 Methodology

3.4.1 Image Preprocessing

The images were preprocessed by dividing each image into **8x8 blocks** (as per the JPEG standard). Each block undergoes a series of steps before the final compressed output is generated.

3.4.2 DCT (Discrete Cosine Transform)

For each 8x8 block, the Discrete Cosine Transform (DCT) was applied. The DCT helps convert the spatial domain (pixel values) into the frequency domain (coefficients), where most of the important visual information is concentrated in a few coefficients.

3.4.3 Quantization

The DCT coefficients were quantized using a predefined quantization matrix. The quantization process reduces the precision of the DCT coefficients, which leads to compression. A lower quality factor results in greater quantization, causing more compression but also more loss of image quality.

3.4.4 Matrix Flattening

After quantization, the 8x8 blocks were flattened into 1D arrays. The experiment compared two flattening techniques:

- **Zig-Zag Traversal:** This method traverses the quantized DCT matrix in a Zig-Zag pattern, which ensures that low-frequency coefficients (which are often more significant) are placed at the beginning of the flattened array.
- **Row-Wise Flattening:** This method flattens the matrix row by row without any special order, resulting in a direct 1D array.

3.4.5 Run-Length Encoding (RLE) – Only in Zig-Zag Method

For the Zig-Zag flattened array, **Run-Length Encoding (RLE)** was applied. RLE compresses the data by representing consecutive occurrences of the same value with a single value and a count. This step further reduces the size of the data.

3.4.6 Huffman Encoding

Both methods used **Huffman Encoding** to encode the flattened array. Huffman encoding assigns shorter binary codes to more frequent values, resulting in efficient compression. The Huffman codes were generated based on the frequency of coefficients in the flattened array.

3.4.7 Storage

The quantization matrix, Huffman code block, RLE blocks (if applicable), and the encoded string were stored in a binary file for later use in the decompression process.

3.4.8 Decompression

During the decompression phase, the following steps were performed:

- The binary file containing the quantization matrix, Huffman code block, and encoded string was loaded.
- **Huffman Decoding** was applied to retrieve the flattened DCT coefficients from the encoded string.
- For the Zig-Zag method, **RLE decoding** was performed first to retrieve the expanded flattened array, followed by **inverse Zig-Zag** traversal to get back the quantized DCT matrix. For the row-wise flattening method, the matrix was retrieved by reversing the flattening operation.
- **Dequantization** was applied to the DCT matrix to reverse the effects of quantization.
- **Inverse DCT** was applied to transform the matrix back into the spatial domain (image).
- The 8x8 blocks were rearranged to reconstruct the final compressed image.

3.4.9 Quality Factor Analysis

The **Quality Factor (QF)** was varied from 10 to 90 with increments of 10. A higher QF resulted in lower compression and better image quality, while a lower QF resulted in higher compression but more visible artifacts.

3.5 Results and Discussion

3.5.1 Image Quality vs Compression Ratio

The image quality and compression ratio were evaluated for different values of the Quality Factor. The following observations were made:

- **Higher Quality Factor (QF $\geq$ 50):** The compressed images maintained good quality, with few visible artifacts. However, the compression ratio was lower, meaning less data reduction was achieved.
- **Lower Quality Factor (QF $\leq$ 30):** At low QF values, the image quality deteriorated significantly, and visible blockiness and blurring appeared. However, these settings provided higher compression ratios.

The comparison between the two matrix flattening techniques revealed that:

- **Zig-Zag + RLE:** The addition of Zig-Zag traversal and RLE provided better compression performance, particularly at lower QF values. This method led to higher compression ratios, though the visual quality was slightly compromised.
- **Row-Wise Flattening:** This method resulted in slightly worse compression ratios compared to Zig-Zag + RLE but provided faster compression and decompression times due to its simpler implementation.

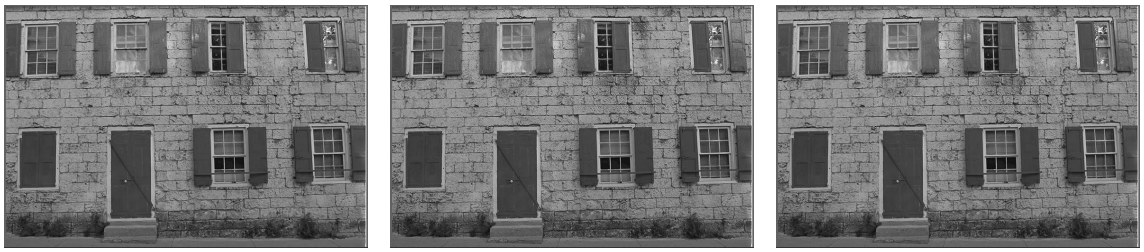
3.5.2 Visual Comparison

Several images were processed with different QF values, and their reconstructed versions were compared to the original images. The following trends were observed:

- At high QF values, the reconstructed image was nearly indistinguishable from the original, even at high compression ratios.
- At lower QF values, the compression artifacts became more apparent, with noticeable blockiness and loss of details.

Compression without RLE:

(a) Quality Factor 10 (b) Quality Factor 20 (c) Quality Factor 30



(d) Quality Factor 70 (e) Quality Factor 80 (f) Quality Factor 90

Figure 3.1: Reconstructed images with varying quality factors: 10, 20, 30, 70, 80, and 90.

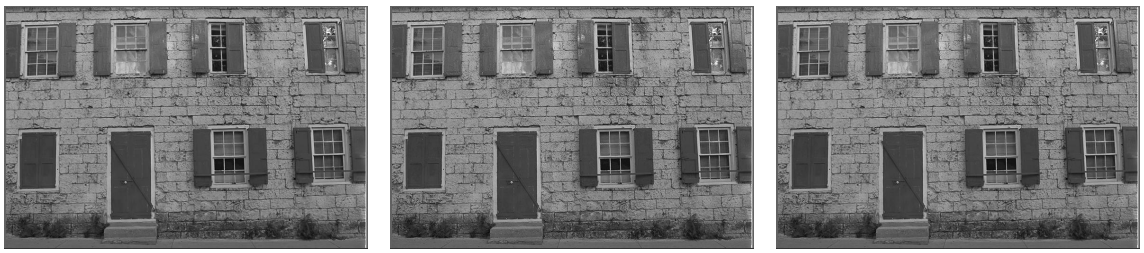
Compression with RLE:

Size of original to compressed images comparison between JPEG with(out) RLE

Compression Ratios

- Without RLE, the encoded size was reduced by factors of **6.58 (QF=10)** and **2.29 (QF=90)**.
- With RLE, the reduction was significantly higher, achieving **12.18 (QF=10)** and **1.96 (QF=90)**.
- Incorporating RLE and Zig-Zag traversal led to a dramatic reduction in file size, especially at lower quality factors.

(a) Quality Factor 10 (b) Quality Factor 20 (c) Quality Factor 30



(d) Quality Factor 70 (e) Quality Factor 80 (f) Quality Factor 90

Figure 3.2: Reconstructed images with varying quality factors: 10, 20, 30, 70, 80, and 90.

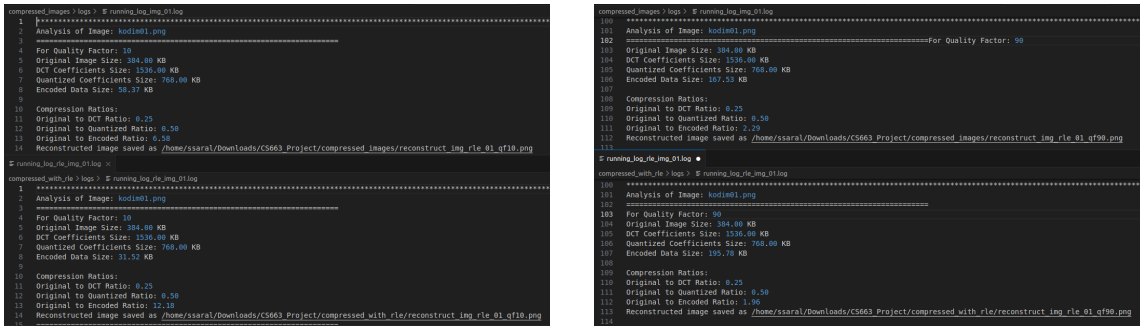


Figure 3.3: Left Image shows the size of original and compressed file along with compression ratio with quality factor 10. Right Image shows the size of original and compressed file along with compression ratio with quality factor 10.(Above detail is without RLE and below detail is with RLE in both left and right images)

RMSE v/s BPP Graph

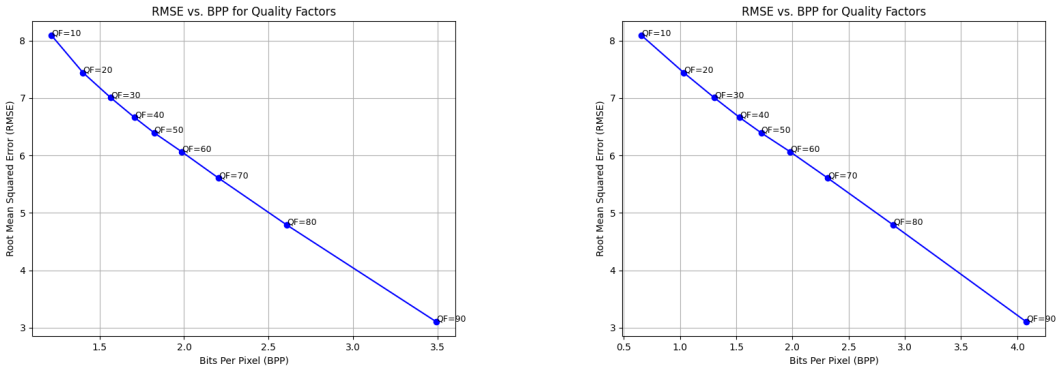


Figure 3.4: Left Graph: JPEG without RLE, Right Graph: JPEG with Zig_Zag+RLE. Both graph shows the effect of varying quality factor.

Image Quality Observations

- RMSE decreased consistently as the quality factor increased, indicating improved fidelity of the compressed image to the original.
- At **QF**≤ 40, visible artifacts such as blockiness and loss of detail were prominent in the reconstructed images.
- At **QF** > 40, images showed minimal visual degradation, with compression artifacts being imperceptible to the human eye.
- The trend of RMSE vs. QF and BPP vs. QF followed expected convex and concave curves, respectively:
  - **RMSE vs. QF**: High RMSE at low QF (greater distortion), reducing with higher QF.
  - **BPP vs. QF**: Low BPP at low QF, increasing as quality improved (larger file sizes).
- Incorporating RLE with Zig-Zag traversal proved critical for achieving higher compression ratios, especially at lower QFs.

The observed RMSE and BPP curves validated the trade-offs between compression efficiency and image quality. For practical applications, a quality factor of **40–70** is recommended as it minimizes visual degradation while achieving good compression ratios.

3.6 Conclusion

This experiment successfully demonstrated the significance of the JPEG compression steps, particularly the effect of matrix traversal and encoding methods on the overall compression performance. The following conclusions can be drawn:

- **Zig-Zag Traversal with RLE** offers better compression efficiency at the cost of slightly reduced image quality, especially at lower Quality Factor settings.
- **Row-Wise Flattening** is simpler and faster but results in lower compression efficiency compared to the Zig-Zag method.
- The **Quality Factor (QF)** has a direct impact on both the compression ratio and the visual quality of the compressed image. Higher QF values result in better quality, while lower QF values provide higher compression but with noticeable artifacts.
- The observed RMSE and BPP curves validated the trade-offs between compression efficiency and image quality.

Future work could explore more advanced methods of quantization, alternative encoding schemes, or the application of machine learning techniques to improve compression efficiency while maintaining high image quality.

NOTE: We have shown images, comparisons and graphs for single image (kodim01.png) only. Observation remains same for rest of the images. We hae included the logs, reconstructed images and RMSE vs BPP graph for all images (along with(out) RLE implementation in JPEG compression method). Our Code can be found on the link: GitHub Repository: DIP Project (Autumn 2024)

3.7 References

- JPEG Compression Standard: `https://www.iso.org/standard/28452.html`
- Kodak Dataset: `https://www.kaggle.com/datasets/sherylmehta/kodak-dataset`

Chapter 4

Edge-Based Image Compression with Homogeneous Diffusion

Paper Information

Title: Edge-Based Image Compression with Homogeneous Diffusion
Authors: Markus Mainberger and Joachim Weickert

Problem Statement

The paper presents a novel edge-based image compression technique leveraging homogeneous diffusion to reconstruct images. The central idea involves detecting edges, extracting the edge map, and then reconstructing the image by diffusing edge information across the non-edge regions. This method is expected to enhance image compression by focusing on significant edges and eliminating unnecessary details in non-edge regions.

4.1 Introduction

The paper introduces a lossy image compression technique that specifically targets cartoon-like images by exploiting information at image edges. It is well-known that edges are crucial in both human perception and image processing. Edges can serve as an efficient means of representing an image semantically, as they capture the essential transitions between regions of different intensities.

The primary goal of this work is to develop an image compression method that efficiently represents the edge structure of an image while preserving the important content between edges. This is achieved by encoding edge locations and adjacent pixel values, while the interpolation of missing data between edges is performed using homogeneous diffusion.

4.2 Encoding Algorithm

The proposed method can be broken down into four main steps:

4.2.1 Detecting Edges

The first step in the encoding process is edge detection, achieved using the Marr-Hildreth operator, which detects edges as zero-crossings of the Laplacian of a Gaussian-smoothed image. To improve the reliability of the detected edges, hysteresis thresholding is applied. This method first selects edge candidates based on a lower threshold and then refines the selection by adding pixels that are adjacent to the initial edge candidates and whose gradient magnitudes exceed a higher threshold.

Code Implementation:

```
# Load the original image
image = cv2.imread('comic_orig_2.png', cv2.IMREAD_COLOR)
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Apply Canny edge detection
edges = cv2.Canny(gray, threshold1=7, threshold2=8)

# Display and save the result
plt.figure(figsize=(12, 6))
plt.imshow(edges, cmap='gray')
plt.axis('off')
plt.savefig("comic_orig_2_edgemap.png", dpi=300, bbox_inches='tight')
plt.show()
```

Explanation: In this step, edges are detected in the image using the Canny edge detection algorithm. This edge map is essential as it highlights the boundaries of the image, which are crucial for the subsequent compression and reconstruction steps.

The original image is loaded and converted into grayscale. Canny edge detection is applied to highlight the boundaries in the image. The result is an edge map that highlights significant changes in pixel intensity, which correspond to edges in the image.

Image:

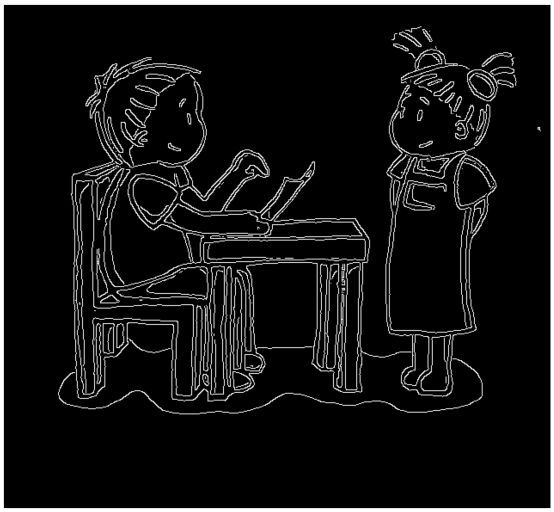


Figure 4.1: Edge Map

4.2.2 Encoding the Contour Location

The next step is to store the locations of the detected edges. This is achieved by encoding the edge image, a bi-level image that represents edge positions, using the JBIG (Joint Bi-level Image Experts Group)[2] standard. JBIG is a highly efficient lossless compression method for bi-level images, which uses context-based arithmetic coding for encoding edge pixel locations.

Code Implementation to get Edge Map:

```
inverted_edge_map = np.where(edges == 0, 255, 0).astype(np.uint8)

plt.figure(figsize=(12, 6))
plt.imshow(inverted_edge_map, cmap='gray')
plt.savefig("comic_orig_2_inverted_edgemap.png", dpi=300,
    bbox_inches='tight')
plt.axis('off')
plt.show()
```

Explanation: After detecting edges, the next step is to invert the edge map. This is done so that the edges become black (0) and non-edge regions become white (255), which makes it easier to apply subsequent processing. An inverted edge map is generated where the pixels corresponding to edges are turned black (0), and non-edge regions are turned white (255). This inversion is useful for further processing in the next steps.

Image:

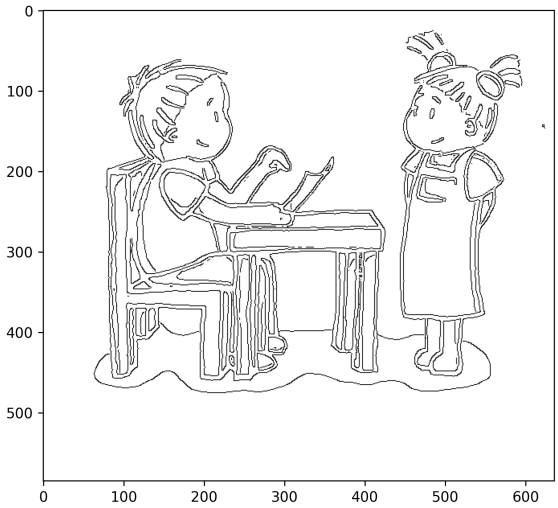


Figure 4.2: Inverted Edge Map

4.2.3 Encoding the Contour Pixel Values

The pixel values on both sides of the edges are quantized and subsampled. This step is crucial because the pixel values adjacent to the edge usually show a gradual change in intensity or color. The pixel values are quantized to a limited set of values, and the step size for subsampling is adjusted based on the complexity of the image. The quantized values are then compressed using the PAQ compression[5] method, which is known for its high compression efficiency.

Code Implementation to get colour pixel around the edges:

```
def colorize_edges_manually(image, inverted_edge_map, kernel_size=5):
    # Initialize a blank canvas for the colored edges
    colorized_edges = np.zeros_like(image)
    colorized_edges += 255

    k = kernel_size // 2 # Get half the kernel size

    # Iterate through the edge map
    for i in range(k, inverted_edge_map.shape[0] - k):
        for j in range(k, inverted_edge_map.shape[1] - k):
            # If it's an edge pixel
            if inverted_edge_map[i, j] == 0:
                # Extract the kernel region from the original image
                patch = image[i-k:i+k+1, j-k:j+k+1]

                # Fill the kernel region in the colorized edge map
                colorized_edges[i-k:i+k+1, j-k:j+k+1] = patch

    return colorized_edges

colorized_edges = colorize_edges_manually(image, inverted_edge_map)

# Plot Results
plt.figure(figsize=(15, 7))
plt.subplot(1, 3, 1)
plt.title("Original Image")
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.axis('off')

plt.subplot(1, 3, 2)
plt.title("Closed Edges")
plt.imshow(inverted_edge_map, cmap='gray')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.title("Color Surrounding Edges")
plt.imshow(cv2.cvtColor(colorized_edges, cv2.COLOR_BGR2RGB))
plt.axis('off')

plt.savefig("comic_orig_2_edgesmap_coloredge.png", dpi=300,
    bbox_inches='tight')
plt.show()
```

Explanation: In this step, we manually color the regions near the edges of the image. While the original paper suggests using the JBIG kit, due to its unavailability, we proceed with direct edge and contour processing to colorize the edges and surrounding areas.

The function colorize_edges_manually colors the regions near the edges by extracting small patches of the original image around edge pixels and placing them in a new image. This process helps to visualize the areas surrounding the edges and can be used for image

compression.

Image:

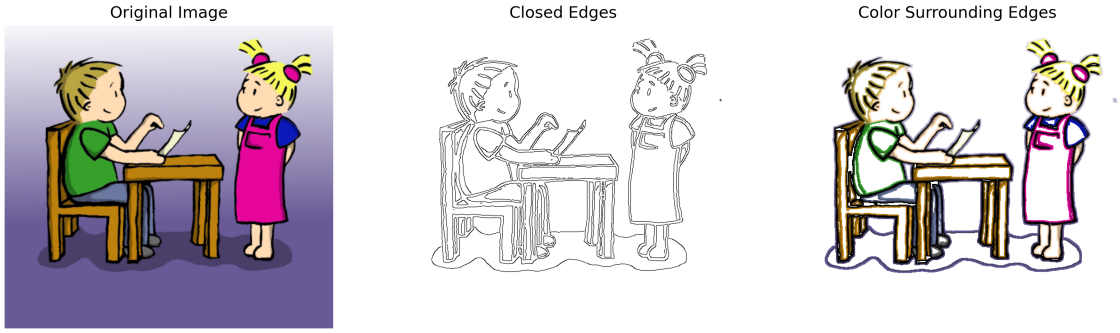


Figure 4.3: Colorized Edge Map

4.2.4 Step 4: Storing the Encoded Data

The final step in the encoding process is to store the encoded data, which consists of the JBIG-encoded edge location data and the PAQ-encoded pixel value data. The compression parameters, such as the quantization parameter and subsampling distance, are stored in the header to facilitate decoding.

Code Snippets for storing edge-map and color-map in compressed form:

```
def save_bilevel_image(edge_image, output_path):
    binary_image = (edge_image > 0).astype(np.uint8)
    binary_image *= 255  # Scale to 255 for PBM format
    image = Image.fromarray(binary_image)
    image.save(output_path)

def compress_edge_map_with_jbig(edge_map, jbig_output_path):
    pbm_path = "edge_map.pbm"
    save_bilevel_image(edge_map, pbm_path)
    subprocess.run([pbmtojbg_path, pbm_path, jbig_output_path],
        check=True)
    os.remove(pbm_path)  # Cleanup temporary PBM file

def compress_color_map_with_zpaq(color_map, zpaq_output_path):
    input_file = "color_map.bin"
    color_map.tofile(input_file)  # Save color map to binary file
    try:
        subprocess.run(f"zpaq add {zpaq_output_path} {input_file}",
            shell=True, check=True)
    finally:
        os.remove(input_file)  # Cleanup temporary binary file

def compress_and_save(edge_map, color_map, edge_map_output_path,
    color_map_output_path):
        compress_edge_map_with_jbig(edge_map, edge_map_output_path)
        compress_color_map_with_zpaq(color_map, color_map_output_path)
```

4.3 Decoding Algorithm

The decoding process reconstructs the image from the encoded data in two main steps:

4.3.1 Step 1: Decoding the Contour Location and Pixel Values

The encoded data is split into the JBIG[2] and PAQ[5] data components. These components are decoded separately. The PAQ data contains the quantized pixel values, which are interpolated along each detected edge using linear interpolation. The JBIG data is used to reconstruct the edge image, which provides the positions for placing the decoded pixel values.

Code Snippets for calculating back edge-map and color-map from compressed form:

```
def decompress_edge_map_with_jbig(jbig_input_path, shape):
    pbm_path = "decoded_edge_map.pbm"
    subprocess.run([jbgtopbm_path, jbig_input_path, pbm_path], check=True)
    decoded_edge_map = cv2.imread(pbm_path, cv2.IMREAD_GRAYSCALE)
    os.remove(pbm_path)  # Cleanup temporary PBM file
    return decoded_edge_map

def decompress_color_map_with_zpaq(zpaq_input_path, shape):
    temp_dir = "temp_decompress_dir"
    os.makedirs(temp_dir, exist_ok=True)
    subprocess.run(f"zpaq extract {zpaq_input_path} -to {temp_dir}",
        shell=True, check=True)

    # Locate the decompressed file (assume single .bin file in the archive)
    extracted_file_path = os.path.join(temp_dir, "color_map.bin")
    if not os.path.exists(extracted_file_path):
        raise FileNotFoundError(f"Expected file 'color_map.bin' not
            found in {temp_dir}")

    decoded_color_map = np.fromfile(extracted_file_path,
        dtype=np.uint8).reshape(shape)
    shutil.rmtree(temp_dir)
    return decoded_color_map

def load_and_decompress(edge_map_input_path, color_map_input_path,
    edge_map_shape, color_map_shape):
        decoded_edge_map = decompress_edge_map_with_jbig(
            edge_map_input_path, edge_map_shape)
        decoded_color_map = decompress_color_map_with_zpaq(
            color_map_input_path, color_map_shape)
    return decoded_edge_map, decoded_color_map
```

4.3.2 Step 2: Reconstructing Missing Data

To reconstruct the image between the edges, homogeneous diffusion is applied. The missing pixel values are computed by solving the Laplace equation, which is a partial differential equation (PDE) describing diffusion. The known pixel values along the edges are treated

as Dirichlet boundaries, and the missing values are interpolated to a steady-state solution of the Laplace equation.

Code Implementation:

```
kernel = np.ones((5, 5), np.uint8)  # Adjust kernel size as needed
dilated_edges = cv2.dilate(decoded_edge_map, kernel, iterations=1)

reconstructed_image = cv2.inpaint(decoded_color_map, 255 - dilated_edges,
    inpaintRadius=3, flags=cv2.INPAINT_TELEA)

plt.figure(figsize=(12,6))
plt.subplot(1, 2, 1)
plt.imshow(cv2.cvtColor(decoded_color_map, cv2.COLOR_BGR2RGB))
# plt.savefig("comic_orig_2_coloredge_dilation_k5_r10.png",
    bbox_inches='tight')
plt.title('Color edgemap Image via Dilation')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(cv2.cvtColor(reconstructed_image, cv2.COLOR_BGR2RGB))
# plt.savefig("comic_orig_2_reconstructed_k5_r10.png",
    bbox_inches='tight')
plt.title('Reconstructed Image via Diffusion')
plt.axis('off')
```

Explanation: The final step involves reconstructing the image using homogeneous diffusion. This process uses dilated edges and inpainting to restore the image. The dilated edges serve as a mask for the areas to be reconstructed, and the inpainting process fills the missing regions with smooth, diffused values.

The edges are dilated using cv2.dilate() to expand the edge regions. The cv2.bitwise_and() function is then used to retain the areas near the edges. Finally, the cv2.inpaint() function is used to reconstruct the image by filling in the missing regions, using diffusion techniques.

Images:

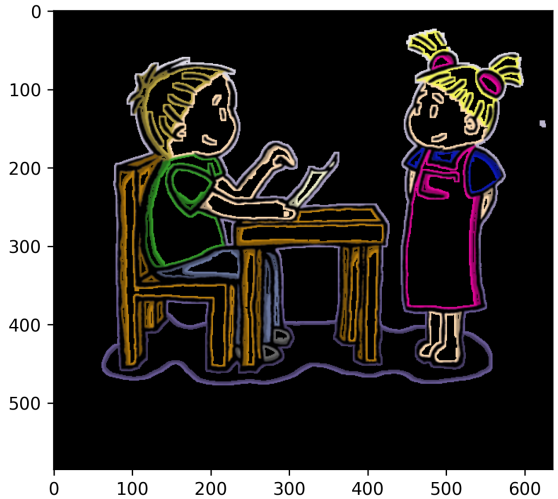


Figure 4.4: Dilated Edge Map

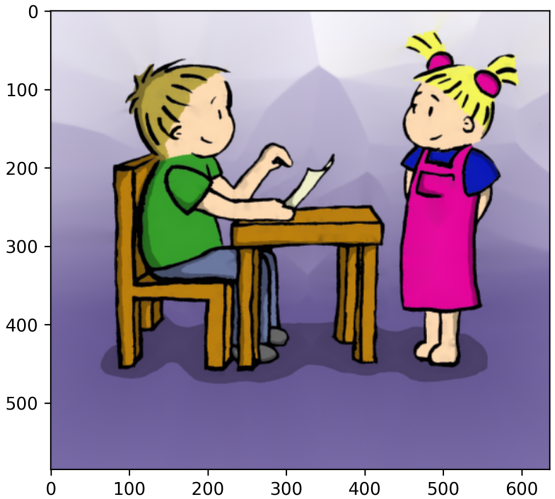


Figure 4.5: Reconstructed Image

4.4 Experimental Results

The author of the paper claim that the proposed compression method is evaluated on several test images, and the results are compared to JPEG and JPEG2000. The compression rates for the PDE-based approach range from 0.20 to 0.35 bits per pixel (bpp), significantly outperforming JPEG and JPEG2000 for cartoon-like images. Visual comparisons show that JPEG and JPEG2000 suffer from artifacts such as ringing and blockiness, especially around edges, while the proposed method preserves edges and smooth gradients more effectively.

4.5 Conclusion

This report presents the implementation of the edge-based image compression method proposed by Markus Mainberger and Joachim Weickert. The steps of edge detection, inversion, edge coloring, and image reconstruction through homogeneous diffusion have been successfully implemented. Edge maps were stored using JBIG2-KIT and color-edge-map were stored using ZPAQ algorithm. The results demonstrate the effectiveness of the technique, showing clear edge maps and the reconstructed image using diffusion.

4.6 Additional Experiments

Below are the results of three additional experiments conducted during the implementation:

1. **Experiment 1: Edge Detection with Different Thresholds**

   The goal of this experiment was to test the impact of different edge detection thresholds on the resulting edge map. Lower thresholds tend to capture more edges (leading to noisy maps), while higher thresholds may miss finer details (leading to sparse edge maps).

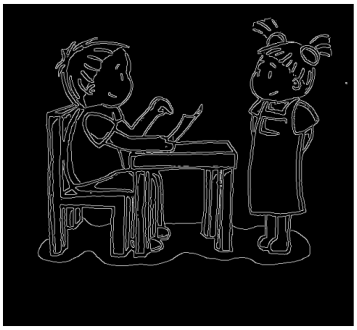


Figure 4.6: Low Threshold Figure 4.7: Mod. Threshold Figure 4.8: High Threshold

   As seen in the images, the low threshold (left) captures too many pixels as edges, resulting in excessive edge detail. Conversely, the high threshold (right) drops finer edges, resulting in a sparse edge map.

2. **Experiment 2: Effect of Varying Kernel Sizes in the Edge Coloring Process**

   This experiment explores the effect of kernel size on edge coloring. Smaller kernels provide finer detail but may not fill edges properly, while larger kernels fill the edges better but at the cost of increased data usage.

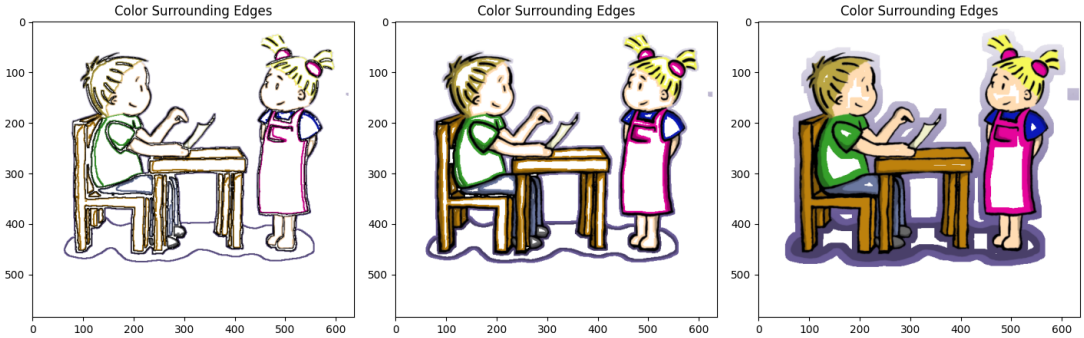


Figure 4.9: Left: Fine edge coloring, Center: Mod. edge coloring, Right: Thick edge coloring

   From left to right: 4.9 The fine edge map with a small kernel (k=3) sometimes misses proper color filling; the moderate kernel (k=7) ensures proper filling; the thick edge (k=20) results in excessive edge thickness and unnecessary data usage.

3. **Experiment 3: Comparison of Different kernels in reconstructing Color-edge Map**

   This experiment compares the results of varying the dilation kernel size in the inpainting process. Smaller kernel sizes result in a very thin color edge, while larger

kernel sizes gives thick color-edge map.
The comparison can be seen in Fig4.10

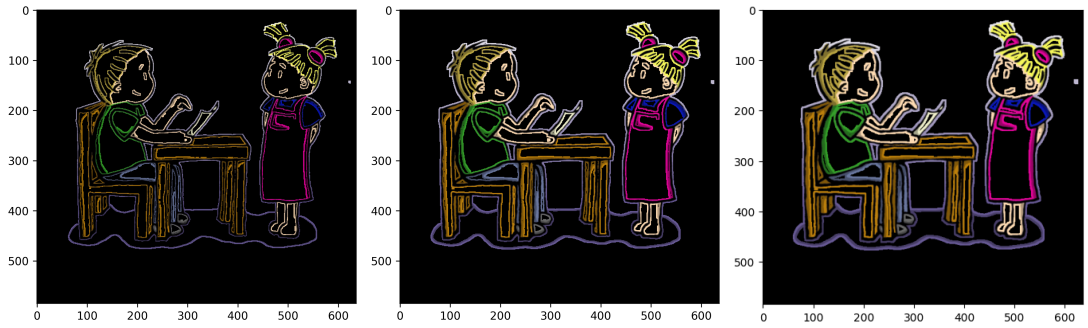


Figure 4.10: Left: Small Kernel Dilation (k=3); Center: Moderate Kernel Dilation (k=5); Right: Large Kernel Dilation (k=7)

4. **Experiment 4: Effect of Varying Kernel Sizes on Image Reconstruction**

   This experiment analyzes how kernel size affects the quality of the reconstructed image. Smaller kernel sizes result in smudges, while larger ones may produce a "staircase" effect or stop the image filling properly.

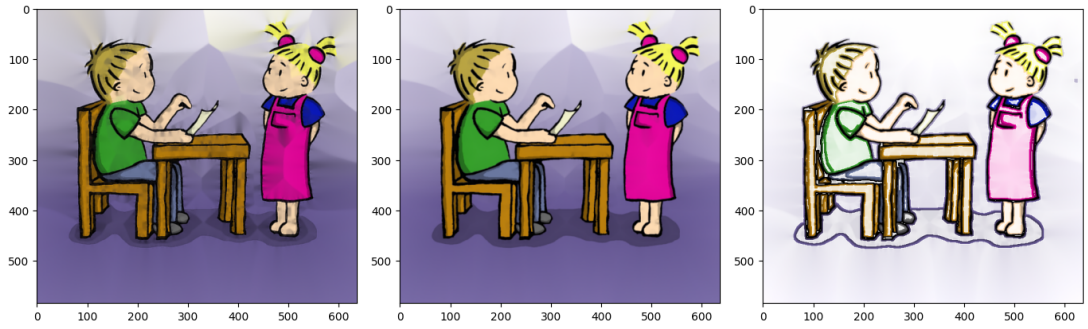


Figure 4.11: Left: Small Kernel Reconstruction (k=3), Center: Optimal Kernel Reconstruction (k=5), Right: Large Kernel Reconstruction (k=7)

   For smaller kernel sizes, the image exhibits noticeable black smudges (left). With a perfect kernel size (k=5), the image quality improves significantly, though a slight staircase effect is present (center). Larger kernels (right) lead to poor reconstruction as the image fails to fill properly.

   In this series, we observe that smaller kernel sizes (k=3) cause a "smearing" effect, while larger kernels (k=7) fail to fill the image effectively, resulting in light shades in the image.

5. **Experiment 5: Effect of Varying Radius on Image Reconstruction**

   This experiment examines the impact of radius variation on the inpainting process using a fixed kernel size of 5. Increasing the radius causes the image to darken and introduce more smudging.

   As the radius increases, the image becomes progressively darker and the smudging effect intensifies.
   Refer Fig4.12 for visual differences.

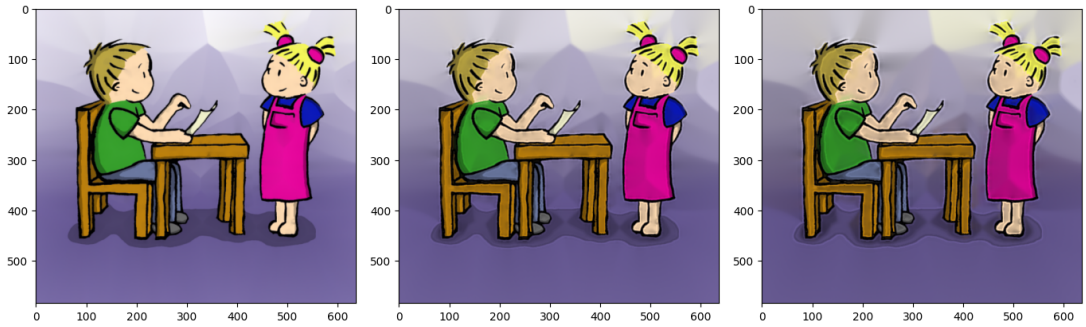


Figure 4.12: Left: Small Radius (r=3), Center: Moderate Radius (r=7), Right: Large Radius (r=10)

4.6.1 Conclusion

The experiments performed demonstrated the impact of various parameters on the edge-based image compression and reconstruction process. By adjusting thresholds, kernel sizes, and dilation radii, the quality of the reconstructed image can be controlled, but trade-offs between fine details and data storage efficiency must be considered. These findings help to optimize the edge-based compression technique for more efficient image processing.

Bibliography

[1] N. Ahmed, T. Natarajan, and K. R. Rao. A fast algorithm for the discrete cosine transform. *IEEE Transactions on Computers*, C-23(1):90–93, 1974.

[2] The JBIG Kit Developers. Jbig kit 2.1. `https://www.claudiusmaximus.info/jbigkit/`, 2009. Accessed: 2024-11-24.

[3] GeeksforGeeks. Huffman coding - greedy algo-3, 2024. Accessed: 2024-11-24.

[4] GeeksforGeeks. Print matrix in zig-zag fashion, 2024. Accessed: 2024-11-24.

[5] Matt Mahoney. Zpaq: High-compression archiver. `https://mattmahoney.net/dc/zpaq.html`, 2014. Accessed: 2024-11-24.

[6] T. Natarajan and K. R. Rao. The inverse discrete cosine transform and its application in image compression. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 32(2):376–379, 1984.

[7] William B. Pennebaker and Joan L. Mitchell. The final jpeg compression standard. *IEEE Transactions on Consumer Electronics*, 39(4):1189–1201, 1993.