MULTIMEDIA SYSTEMS & NETWORKS
(ELE4410)

Assignment

Topic: Lossless Image Compression

Submitted by

Name: Sanskriti
Faculty Number:20ELB270
Enrolment No: GK8667

DEPARTMENT OF ELECTRONICS ENGINEERING

ZAKIR HUSAIN COLLEGE OF ENGINEERING & TECHNOLOGY

ALIGARH MUSLIM UNIVERSITY ALIGARH

CONTENT

ABSTRACT:

This assignment provides a focused exploration of image lossless compression specifically using Huffman coding. Huffman coding is a variable-length, prefix-free code that efficiently reduces image file sizes while preserving all original data. The assignment begins by discussing the need for lossless compression in applications such as medical imaging and data archival where data integrity is paramount.

A detailed explanation of Huffman coding is presented, including its principles of frequency analysis, code assignment, and prefix-free properties. Through an illustrative example, the assignment demonstrates how Huffman coding optimally assigns shorter codes to more frequent symbols, resulting in an efficient compressed bitstream.

Real-world applications of Huffman coding in image compression are highlighted, including its historical use in fax machines, its incorporation into Lossless JPEG (JPEG-LS), and its role in the Portable Network Graphics (PNG) image format.

In conclusion, this assignment emphasizes the importance and effectiveness of Huffman coding in lossless image compression. Understanding Huffman coding empowers users to reduce image file sizes without compromising data integrity. Future research directions could explore optimizations and enhancements to Huffman coding techniques for even more efficient compression.

This focused exploration aims to provide a comprehensive understanding of image lossless compression using Huffman coding, offering practical insights for applications in various fields requiring high-quality image storage and transmission.

INTRODUCTION

In the digital era, image files often occupy considerable space due to their detailed and high-resolution nature. Image compression techniques are essential to reduce file sizes for efficient storage, transmission, and processing. These techniques can broadly be categorized into two categories: lossless and lossy compression.

Lossless Image Compression Techniques

Lossless compression aims to reduce file size without losing any image data. This is crucial in applications where maintaining pixel-perfect accuracy is vital. Here are some common lossless compression techniques:

1. **Huffman Coding**

Huffman coding, as discussed earlier, assigns shorter codes to more frequent symbols and longer codes to less frequent ones. This technique is highly efficient in reducing redundancy in image data.

2. **Lempel-Ziv-Welch (LZW)**

LZW is another popular algorithm for lossless compression. It works by replacing sequences of symbols with codes. As the algorithm encounters repeated sequences, it replaces them with shorter codes, thereby reducing file size.

3. **Run-Length Encoding (RLE)**

RLE is a simple yet effective technique that replaces consecutive identical pixels with a count and a single pixel value. This is particularly useful in images with large areas of uniform color.

4. **Arithmetic Coding**

Arithmetic coding is a more complex method that replaces symbols with fractional values. It achieves high compression ratios by assigning shorter codes to more probable symbols.

Lossy Image Compression Techniques

Lossy compression sacrifices some image quality to achieve higher compression ratios. This is acceptable in scenarios where a slight loss of fidelity is tolerable. Common lossy compression techniques include:

1. **Discrete Cosine Transform (DCT)**

DCT is the foundation of the JPEG image compression standard. It converts spatial image data into frequency components, allowing for high compression by discarding high-frequency components that the human eye is less sensitive to.

2. **Wavelet Transform**

Wavelet Transform is used in JPEG 2000 and other formats. It provides a multiresolution representation of images, allowing for selective compression of different frequency bands.

3. **Transform Coding**

Transform coding, like DCT, transforms image data into a different domain where compression is more efficient. It allows for discarding less critical information while preserving essential details.

HUFFMAN CODING

Huffman coding, developed by David A. Huffman in 1952, is a variable-length, prefix-free code used for lossless data compression. It takes advantage of the non-uniform distribution of symbols in most data sources, including images.

The key principles include:

Frequency Analysis: Huffman coding begins with analyzing the frequency of symbols (like pixel values in an image) in the data to be compressed.

Code Assignment: More frequent symbols are assigned shorter codes, while less frequent symbols receive longer codes. This optimizes the efficiency of the encoding process.

Prefix-Free: The resulting codes are prefix-free, meaning no code is a prefix of another code. This property ensures unambiguous decoding.

Encoding Process:

The encoding process involves building a Huffman tree based on the symbol frequencies. This binary tree has leaf nodes representing symbols and their corresponding codes. The steps include:

Frequency Analysis: Calculate the frequency of each symbol in the image data.

Build Huffman Tree: Construct a binary tree where each leaf represents a symbol and its frequency, merging nodes with the lowest frequencies until a single root node remains.

Assign Codes: Traverse the tree to assign codes, with left branches representing '0' and right branches representing '1'.

Generate Encoded Bitstream: Replace each symbol with its corresponding Huffman code to create the compressed bitstream.

Decoding Process

Decoding the compressed bitstream involves traversing the Huffman tree in a similar manner. The steps are:

Start at Root: Begin at the root of the Huffman tree.

Traverse Tree: For each bit read from the compressed stream, follow the corresponding path in the tree. When a leaf node is reached, output the corresponding symbol.

Repeat: Continue this process until the entire compressed stream is decoded.

Example: Huffman Coding in Action:

Let's consider a simple example to illustrate Huffman coding's efficiency. Suppose we have an image with the following pixel frequencies:

White Pixels: 3000

Black Pixels: 1500

Gray Pixels: 500

Other Pixels: 100

Applying Huffman coding to this data, we would assign shorter codes to white and black pixels, followed by gray pixels, and longer codes to the less frequent other pixels. This results in a compressed bitstream that represents the image with minimal redundancy.

Applications:

Fax Machines

Huffman coding found significant use in fax machines for transmitting images over telephone lines. The efficiency of Huffman coding in compressing black-and-white images made it ideal for this purpose, where reducing file size was crucial for speedy transmission.

Lossless JPEG (JPEG-LS)

The Lossless JPEG standard, also known as JPEG-LS, utilizes Huffman coding in its baseline sequential mode. This mode provides lossless compression for images while maintaining compatibility with the JPEG format. It employs Huffman tables to encode image data efficiently.

PNG Image Format

The Portable Network Graphics (PNG) image format, renowned for its lossless compression, incorporates Huffman coding in its compression algorithm. PNG uses a combination of deflate compression (which includes Huffman coding) and adaptive filtering to achieve excellent compression ratios for a wide range of images.

Implementation of Image Compression Using Huffman Coding Techniques

Step 1: Open and Read Image as Binary

The first step is to open and read the image as binary and store it in a string.

Step 2: Create Frequency Dictionary

The second step is to separate the binary string into 8-bit chunks and create a frequency dictionary for each byte.

Step 3: Build Huffman Tree and Generate Huffman Codes

Next, we import the 'Heapq' library to use a min-heap and build a Huffman tree. We then generate Huffman codes for each byte.

Step 4: Replace Huffman Codes in Binary String for Compression

In this step, we replace the Huffman codes in the original binary string to compress the data.

Step 5: Decompression - Reversing the Process

For the last part, we implement the reverse process to decompress the image using the Huffman codes and Huffman tree.

Algorithm Huffman Coding

Input: Image file path (input_file_path)
Output: Compressed file ('compressed_file'), Decompressed file ('decompressed_file'), Compression Ratio (CR)

1. Read Image File and Convert to Binary String
   - Open the image file at input_file_path in binary read mode.
   - Read the image file byte by byte.
   - Convert each byte to a binary string of 8 bits.
   - Append these binary strings to create a single bit_string.
   - Close the image file.

2. Calculate Frequencies of 8-bit Chunks
   - Initialize an empty dictionary 'frequencies' to store frequencies.
   - Iterate through bit_string:
     - Take 8-bit chunks and check if they exist in 'frequencies':
       - If exists, increment the frequency by 1.
       - If not, add the chunk to 'frequencies' with frequency 1.
   - Sort 'frequencies' by values (ascending order) and store as 'sorted_frequencies'.

3. Build Huffman Tree
   - Initialize an empty list 'nodes' to store node objects.
   - For each item in 'sorted_frequencies':
     - Create a new node object with:
       - Frequency from 'sorted_frequencies'.
       - Symbol as the 8-bit chunk.
       - Set 'huff' as an empty string.
     - Push this node object into 'nodes' using heapq.
   - While 'nodes' has more than one node:
     - Pop two nodes with the lowest frequencies as 'left' and 'right'.
     - Assign 'huff' as 0 to 'left' and 1 to 'right'.
     - Create a new node 'newNode' with:
       - Frequency as the sum of 'left' and 'right' frequencies.

- Symbol as the concatenation of 'left' and 'right' symbols.
- Left child as 'left' and right child as 'right'.
- Push 'newNode' into 'nodes' using heapq.

4. Generate Huffman Codes
- Define a recursive function 'printNodes(node, val='')':
- Takes a node and a string 'val' as parameters.
- Concatenate 'node.huff' to 'val'.
- If 'node' has a left child, recursively call 'printNodes(node.left, newVal)'.
- If 'node' has a right child, recursively call 'printNodes(node.right, newVal)'.
- If 'node' is a leaf node (no left or right child), print 'node.symbol' and its Huffman code.
- Store each 'node.symbol' and its Huffman code in the global 'huffman_codes' dictionary.
- Call 'printNodes' with the root node from the Huffman tree.

5. Compress Image Data
- Initialize an empty string 'bit_string_out' for compressed bits.
- Iterate through bit_string:
- Take 8-bit chunks and get their Huffman codes from 'huffman_codes'.
- Append the Huffman code of each chunk to 'bit_string_out'.

6. Convert Compressed Bits to Bytes
- Create an empty 'output' bytearray to store compressed bytes.
- Convert 'bit_string_out' to bytes:
- Iterate through 'bit_string_out' in chunks of 8 bits.
- Convert each 8-bit chunk to an integer and append to 'output' as a byte.

7. Write Compressed Data to File
- Open a new binary write file ('compressed_file') to write compressed data.
- Write the 'output' bytearray to 'compressed_file'.
- Close 'compressed_file'.

8. Calculate Compression Ratio (CR)
- Calculate Compression Ratio (CR) as:

- CR = Original Image File Size / Compressed File Size
- Print "CR: ", CR

9. Decompress Data
   - Read the compressed file ('compressed_file') byte by byte.
   - Convert each byte to a binary string.
   - Use the 'huffman_codes' dictionary to find corresponding symbols.
   - Concatenate the symbols to create 'bit_string_out'.

10. Convert Decompressed Bits to Bytes
    - Create an empty 'output' bytearray to store decompressed bytes.
    - Convert 'bit_string_out' to bytes:
      - Iterate through 'bit_string_out' in chunks of 8 bits.
      - Convert each 8-bit chunk to an integer and append to 'output' as a byte.

11. Write Decompressed Data to File
    - Open a new binary write file ('decompressed_file') to write decompressed data.
    - Write the 'output' bytearray to 'decompressed_file'.
    - Close 'decompressed_file'.

12. Calculate File Sizes and Print
    - Get the sizes of the original image file, compressed file, and decompressed file.
    - Print:
      - "Original Image File Size:", original_size, "bytes"
      - "Compressed File Size:", compressed_size, "bytes"
      - "Decompressed File Size:", decompressed_size, "bytes"

13. End of Algorithm

RESULT:

```
CR:  1.0021284304782505
Original Image File Size: 68800 bytes
Compressed File Size: 68654 bytes
Decompressed File Size: 68800 bytes
```

[1]

```
CR:  1.0002828688039793
Original Image File Size: 102108 bytes
Compressed File Size: 102080 bytes
Decompressed File Size: 102108 bytes
```

[2]

```
CR:  1.0005709173491157
Original Image File Size: 454353 bytes
Compressed File Size: 454094 bytes
Decompressed File Size: 454353 bytes
```

[3]

CONCLUSION:

In the realm of image compression, understanding both lossless and lossy techniques is crucial for optimizing file sizes while maintaining image quality. Lossless techniques like Huffman coding, LZW, and RLE excel in scenarios where preserving every detail is essential. On the other hand, lossy techniques such as DCT and wavelet transform offer higher compression ratios at the expense of some image fidelity.

Huffman coding, with its elegant approach of assigning variable-length codes based on symbol frequencies, remains a cornerstone of lossless compression. Its applications in various compression standards and formats highlight its efficiency and versatility.

As technology advances, the quest for more efficient image compression continues. The balance between file size reduction and image quality preservation remains a central focus. By exploring the principles, techniques, and applications of image compression, we pave the way for more effective utilization of digital imagery in diverse fields, from healthcare to entertainment and beyond.

This comprehensive overview provides a deep dive into image compression techniques, from the fundamental principles of Huffman coding to real-world applications across various standards and formats. Understanding these techniques empowers us to make informed decisions when dealing with image data, whether it's for storage, transmission, or processing needs.

REFERENCES:

[1] P. Meerwald, "Lena image," Salzburg University. [Online]. Available: https://www.cosy.sbg.ac.at/~pmeerw/Watermarking/lena.html. Accessed on: Mar. 11, 2024.

[2] "15 facts about hot air balloon festival," Facts.net, Jul. 2023. [Online]. Available: https://facts.net/wp-content/uploads/2023/07/15-facts-about-hot-air-balloon-festival-1690091037.jpg. Accessed on: Mar. 11, 2024.

[3] "Image of a hot air balloon," Bing, [Online]. Available: https://th.bing.com/th/id/R.ab99b6346bc3489f1b52ddfd508dd779?rik=m%2b8ZTXYkUCpsCQ&riu=http%3a%2f%2fclipart-library.com%2fimages%2fATbrgXbjc.jpg&ehk=d7jQyN7yqqaSJNTGdrMl4xfRKOgtYG526wqeeVL0B8U%3d&risl=&pid=ImgRaw&r=0. Accessed on: Mar. 11, 2024.

[4] S. R. Khaitu and S. P. Panday, "Canonical Huffman Coding for Image Compression," 2018 IEEE 3rd International Conference on Computing, Communication and Security (ICCCS), Kathmandu, Nepal, 2018, pp. 184-190, doi: 10.1109/CCCS.2018.8586816.

[5] S. Li, Y. Shen, and M. Li, "Efficient image compression using deep learning," Multimedia Tools and Applications, vol. 80, no. 1, pp. 555-569, Jan. 2021. DOI: 10.1007/s11042-021-11017-5.