Huffman Coding for Image Compression

A MINI PROJECT REPORT

Submitted by

BL.EN.U4AIE21006 Akhilesh P

BL.EN.U4AIE21008 Amal Krishna K

BL.EN.U4AIE21112 S Karthick Bharadwaj

**In partial fulfilment for the award of the degree**

**Of**

**BACHELOR IN TECHNOLOGY**

**IN**

COMPUTER SCIENCE AND ENGINEERING (ARTIFICIAL INTELLIGENCE)



AMRITA SCHOOL OF ENGINEERING, BANGALORE

AMRITA VISHWA VIDYAPEETHAM

BANGALORE – 560035

Abstract

Image compression techniques play a crucial role in reducing the size of digital images while maintaining their visual quality. This project focuses on the implementation of the Huffman coding algorithm for image compression. Huffman coding is a widely used lossless compression algorithm that assigns shorter codes to frequently occurring symbols, resulting in efficient compression.

The project begins by representing the image data in the .bmp format, which supports lossless compression. Pre-processing techniques such as color space conversion, down-sampling, and image resizing could be applied to optimize the compression results based on specific image requirements.

The core of the project revolves around the implementation of the Huffman coding algorithm. The algorithm involves calculating the frequency of each symbol (pixel intensity value) in the image, constructing a Huffman tree using a priority queue, assigning Huffman codes to each symbol, encoding the image data using the generated Huffman codes, and finally compressing the encoded data into a compressed format.

The project is implemented using the java programming language, utilizing libraries such as BufferedReader for image manipulation.

Contents

1. Abstract
2. Huffman Coding
3. Algorithm Analysis
4. Program Flow
5. Code
6. References

Huffman Coding

Huffman coding is a widely used algorithm for data compression, specifically for lossless compression. It was developed by David A. Huffman in the 1950s while he was a student at MIT. Huffman coding achieves compression by assigning shorter codes to more frequently occurring symbols, thereby reducing the overall size of the data without losing any information.

1. Basic Principle:

Huffman coding operates on the principle of variable-length prefix codes, where the length of the code assigned to each symbol is not fixed. The more frequently a symbol occurs, the shorter its code will be, resulting in efficient compression.

2. Algorithm Steps:

The Huffman coding algorithm involves the following steps:

a) Frequency Calculation: The frequency of occurrence of each symbol in the input data is determined. Symbols can be individual characters, pixels in an image, or any other discrete units.

b) Building the Huffman Tree: A binary tree is constructed using a priority queue (e.g., a min-heap) based on the frequencies of symbols. Initially, each symbol is represented as a leaf node in the tree.

c) Merging Nodes: The two nodes with the lowest frequencies are merged into a single node until the entire tree is constructed. The merged node becomes the parent node, with the combined frequency of its children.

d) Assigning Huffman Codes: Starting from the root of the tree, a binary code is assigned to each symbol. Traversing left branches corresponds to appending a '0' to the code, while traversing right branches corresponds to appending a '1'.

e) Encoding: The input data is encoded by replacing each symbol with its corresponding Huffman code. The encoded data is a compressed representation of the original data.

3. Decoding:

To decode the encoded data, the Huffman tree is used. Starting from the root, each bit of the encoded data is read. If a '0' is encountered, the left branch is followed; if a '1' is encountered, the right branch is followed. The process continues until a leaf node is reached, and the corresponding symbol is obtained.

4. Compression Ratio:

Huffman coding achieves compression by representing frequently occurring symbols with shorter codes. As a result, the compressed data is smaller than the original data, reducing storage requirements and facilitating faster transmission.

5. Advantages:

- Lossless Compression: Huffman coding retains all the original data without any loss, making it suitable for applications where preservation of information is crucial.

- Efficiency: The algorithm constructs optimal codes based on symbol frequencies, resulting in efficient compression.

- Simplicity: Huffman coding is conceptually simple and can be implemented using basic data structures.

6. Limitations:

- Variable-Length Codes: While Huffman coding achieves good compression, it introduces variable-length codes, which may require additional bits for encoding certain symbols, leading to slight overhead in storage.

- Encoding Overhead: The compression ratio of Huffman coding may not be as high as other more complex compression algorithms, such as those used in modern image or video codecs.

In conclusion, Huffman coding is a powerful technique for lossless data compression. It assigns shorter codes to frequently occurring symbols, enabling efficient compression without losing any information. By leveraging the principles of variable-length prefix codes, Huffman coding has found widespread application in various fields, including file compression, image compression, and network communication, contributing to the efficient storage and transmission of digital data.

Algorithm Analysis

\*\*Algorithmic Analysis: Space and Time Complexity of Huffman Coding\*\*

Huffman coding is an efficient algorithm for data compression, but it is crucial to analyze its space and time complexity to understand its efficiency in terms of memory usage and computational requirements.

1. Space Complexity:

The space complexity of Huffman coding refers to the amount of memory required by the algorithm to perform its operations. There are several components contributing to the space complexity:

- Input Data: The space complexity depends on the size of the input data that needs to be compressed. If the input data consists of n symbols, the space complexity would be O(n).

- Frequency Calculation: The frequency calculation step requires maintaining a frequency table to count the occurrences of each symbol in the input data. The space required to store this frequency table is proportional to the number of unique symbols in the input data.

- Huffman Tree: The Huffman tree is built using a priority queue (e.g., a min-heap) to merge nodes based on their frequencies. The space complexity of the Huffman tree construction is O(n) since, in the worst case, there can be n-1 internal nodes in the tree.

- Encoding: The space required to store the encoded data depends on the size of the input data and the length of the Huffman codes assigned to each symbol. The space complexity for storing the encoded data is typically O(n) since the length of the encoded data is directly proportional to the size of the input data.

In summary, the overall space complexity of Huffman coding is typically O(n) or proportional to the size of the input data.

2. Time Complexity:

The time complexity of Huffman coding refers to the computational cost or the number of operations required to perform the compression.

- Frequency Calculation: Calculating the frequency of each symbol in the input data takes O(n) time, where n is the size of the input data.

- Building the Huffman Tree: Constructing the Huffman tree involves merging nodes based on their frequencies. Each merge operation takes O(log n) time in the priority queue. Since there can be n-1 merge operations, the overall time complexity for building the Huffman tree is O(n log n).

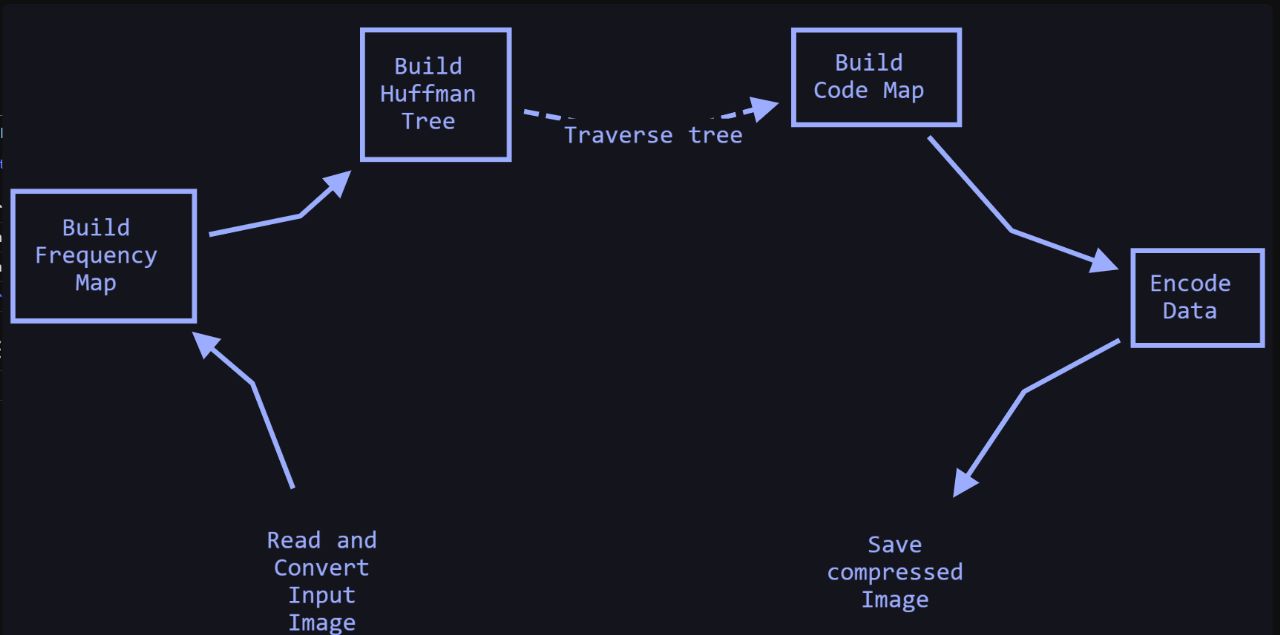
- Assigning Huffman Codes: Assigning Huffman codes to each symbol involves traversing the Huffman tree. The time complexity of assigning codes to a single symbol is proportional to the depth of its corresponding leaf node in the tree. In the worst case, where the tree is unbalanced, the time complexity for assigning codes to all symbols can be O(n).

- Encoding: Encoding the input data using the generated Huffman codes requires iterating through each symbol in the input data and retrieving its corresponding code. The time complexity for encoding the entire input data is O(n).

In conclusion, the overall time complexity of Huffman coding is typically O(n log n) due to the construction of the Huffman tree, while the frequency calculation, code assignment, and encoding operations contribute to a linear time complexity of O(n).

Understanding the space and time complexity of Huffman coding helps in assessing its efficiency and performance characteristics. Despite the relatively higher time complexity compared to some other compression algorithms, Huffman coding's efficient space utilization and the ability to achieve high compression ratios make it a popular choice for various applications involving lossless data compression.

Program Flow



1. The program starts in the `main` method. It prompts the user to enter the path of the input image and the path to save the compressed image.

2. The `compressImage` method is called, which takes the input image path and output path as parameters.

3. Inside `compressImage`, the input image is read using `ImageIO` and converted into an array of bytes. The frequency map is built using the `buildFrequencyMap` method, which counts the occurrences of each pixel value (character) in the image.

4. The Huffman tree is built using the `buildHuffmanTree` method. The method initializes a priority queue (`pq`) with a custom `HuffmanComparator` to compare nodes based on their frequencies. Each character with its frequency is added to the priority queue as a leaf node.

5. The priority queue is then processed to build the Huffman tree. The two nodes with the lowest frequencies are repeatedly removed from the queue, merged into a new parent node, and added back to the queue. This process continues until only one node remains, which becomes the root of the Huffman tree.

6. The code map is built using the `buildCodeMap` method, which assigns a binary code to each character in the Huffman tree. The method uses a recursive helper function `buildCodeMapHelper` to traverse the tree and build the code map.

7. The input image pixels are encoded using the code map. Each pixel is replaced with its corresponding Huffman code obtained from the code map. The encoded data is stored as a `StringBuilder` in the `encodedData` variable.

8. The encoded data is then converted into bytes using the `encodeData` method. It calculates the number of bytes needed to store the encoded data and sets the corresponding bits in the byte array.

9. Finally, the compressed data is saved to the output path using the `saveCompressedImage` method, which writes the byte array to a file.

10. The program completes execution, and the compressed image is saved at the specified output path.

Overall, the flow of control in the code involves reading the input image, building the Huffman tree based on the pixel frequencies, encoding the image pixels using the Huffman codes, and saving the compressed data as a new image file. The code utilizes various helper methods to perform specific tasks, such as building frequency maps, constructing the Huffman tree, assigning codes, and encoding the data.

CODE

import java.awt.image.BufferedImage;

import java.io.\*;

import java.util.\*;

import javax.imageio.ImageIO;

class HuffmanNode implements Comparable<HuffmanNode>

{

int frequency;

char data;

HuffmanNode left, right;

public HuffmanNode(int frequency, char data, HuffmanNode left, HuffmanNode right)

{

this.frequency = frequency;

this.data = data;

this.left = left;

this.right = right;

}

public boolean isLeaf()

{

return left == null && right == null;

}

public int compareTo(HuffmanNode node)

{

return frequency - node.frequency;

}

}

class HuffmanComparator implements Comparator<HuffmanNode>

{

public int compare(HuffmanNode node1, HuffmanNode node2)

{

return node1.frequency - node2.frequency;

}

}

public class HuffmanCompression

{

public static void compressImage(String imagePath, String outputPath)

{

try

{

BufferedImage image = ImageIO.read(new File(imagePath));

ByteArrayOutputStream = new ByteArrayOutputStream();

ImageIO.write(image, "jpg", byteArrayOutputStream);

byte[] pixels = byteArrayOutputStream.toByteArray();

Map<Character, Integer> frequencyMap = buildFrequencyMap(pixels);

HuffmanNode root = buildHuffmanTree(frequencyMap);

Map<Character, String> codeMap = buildCodeMap(root);

StringBuilder encodedData = new StringBuilder();

for (byte pixel : pixels)

{

char pixelChar = (char) (pixel & 0xFF);

encodedData.append(codeMap.get(pixelChar));

}

byte[] compressedData = encodeData(encodedData.toString());

saveCompressedImage(compressedData, outputPath);

}

catch (IOException e)

{

e.printStackTrace();

}

}

private static Map<Character, Integer> buildFrequencyMap(byte[] pixels)

{

Map<Character, Integer> frequencyMap = new HashMap<>();

for (byte pixel : pixels)

{

char pixelChar = (char) (pixel & 0xFF);

frequencyMap.put(pixelChar, frequencyMap.getOrDefault(pixelChar, 0) + 1);

}

return frequencyMap;

}

private static HuffmanNode buildHuffmanTree(Map<Character, Integer> frequencyMap)

{

PriorityQueue<HuffmanNode> pq = new PriorityQueue<>(new HuffmanComparator());

for (Map.Entry<Character, Integer> entry : frequencyMap.entrySet())

{

pq.add(new HuffmanNode(entry.getValue(), entry.getKey(), null, null));

}

while (pq.size() > 1)

{

HuffmanNode left = pq.poll();

HuffmanNode right = pq.poll();

int sumFrequency = left.frequency + right.frequency;

pq.add(new HuffmanNode(sumFrequency, '\0', left, right));

}

return pq.poll();

}

private static Map<Character, String> buildCodeMap(HuffmanNode root)

{

Map<Character, String> codeMap = new HashMap<>();

buildCodeMapHelper(root, "", codeMap);

return codeMap;

}

private static void buildCodeMapHelper(HuffmanNode node, String code, Map<Character, String> codeMap)

{

if (node.isLeaf())

{

codeMap.put(node.data, code);

return;

}

buildCodeMapHelper(node.left, code + "0", codeMap);

buildCodeMapHelper(node.right, code + "1", codeMap);

}

private static byte[] encodeData(String encodedData)

{

int length = encodedData.length();

int numBytes = (length + 7) / 8;

byte[] bytes = new byte[numBytes];

for (int i = 0; i < length; i++)

{

if (encodedData.charAt(i) == '1')

{

int byteIndex = i / 8;

int bitOffset = i % 8;

bytes[byteIndex] |= (1 << (7 - bitOffset));

}

}

return bytes;

}

private static void saveCompressedImage(byte[] compressedData, String outputPath)

{

try (FileOutputStream outputStream = new FileOutputStream(outputPath))

{

outputStream.write(compressedData);

}

catch (IOException e)

{

e.printStackTrace();

}

}

public static void main(String[] args)

{

Scanner = new Scanner(System.in);

System.out.print("Enter the path of the input image: ");

String imagePath = scanner.nextLine();

System.out.print("Enter the path to save the compressed image: ");

String outputPath = scanner.nextLine();

compressImage(imagePath, outputPath);

scanner.close();

}

}

REFERENCES

* Srikanth, Sukadev Meher, April 3-5, 2013, Compression Efficiency for Combining Different Embedded Image Compression Techniques with Huffman Encoding, International conference on Communication and Signal Processing
* Djuned Fernando Djusdek , Hudan Studiawan , and Tohari Ahmad,2016,Adaptive Image Compression Using Adaptive Huffman and LZW, International Conference on Information, Communication Technology and System
* Xiaoxiao Liu ,Ping An , Yilei Chen & Xinpeng Huang , 5 May 2021 , An improved lossless image compression algorithm based on Huffman coding , Shanghai Institute for Advanced Communication and Data Science, School of Communication and Information Engineering, Shanghai University, Shanghai 200444, China
* Asian Journal of Computer Science And Information Technology, "A Fast Fractal Image Compression Using Huffman Coding", 2:9 (2012) 272-275
* Wei-Yi Wei, Graduate Institute of Communication Engineering National Taiwan University, "An Introduction to Image Compression", Taipei, Taiwan, ROC