PROJECT SYNOPSIS

Text File Compression And Decompression Using Huffman Coding

BACHELOR OF TECHNOLOGY

COMPUTER SCIENCE AND ENGINEERING

Submitted by:

Mridul Gupta(20630103)
Shivam Kumar(216303001)
Chandan Kumar Singh(206301082)
Siddharth Sharma(206301112)

Department of CSE, FET Gurukul Kangri Deemed to be University, Haridwar

Introduction

Text files can be compressed to make them smaller and faster to send, and unzipping files on devices has a low overhead. The process of encoding involves changing the representation of a file so that the (binary) compressed output takes less space to store and takes less time to transmit while retaining the ability to reconstruct the original file exactly from its compressed representation. Text files can be of various file types, such as HTML, JavaScript, CSS, .txt, and so on. Text compression is required because uncompressed data can take up a lot of space, which is inconvenient for device storage and file sharing.

The size of the text file can be reduced by compressing it, which converts the text to a smaller format that takes up less space. It typically works by locating similar strings/characters within a text file and replacing them with a temporary binary representation to reduce the overall file size.

How Does The Process Of Compression Work?

The size of the text file can be reduced by compressing it, which converts the text to a smaller format that takes up less space. It typically works by locating similar strings/characters within a text file and replacing them with a temporary binary representation to reduce the overall file size. There are two types of file compression,

- Lossy compression: Lossy compression shrinks a file by permanently removing certain elements, particularly redundant elements.
- Lossless compression: Lossless compression can restore all elements of a file during decompression without sacrificing data and quality.

Text encoding is also of two types:

1. Fixed length encoding and
2. Variable length encoding.

The two methods differ in the length of the codes. Analysis shows that variable-length encoding is much better than fixed-length encoding. Characters in variable-length encoding are assigned a variable number of bits based on their frequency in the given text. As a result, some characters may require a single bit, while others may require two bits, while still others may require three bits, and so on.
Compressing a Text File:

We use the **Huffman Coding** algorithm for this purpose which is a greedy algorithm that assigns variable length binary codes for each input character in the text file. The length of the binary code depends on the frequency of the character in the file. The algorithm suggests creating a binary tree where all the unique characters of a file are stored in the tree’s leaf nodes.

- The algorithm works by first determining all of the file’s unique characters and their frequencies.

•The characters and frequencies are then added to a Min-heap.

•It then extracts two minimum frequency characters and adds them as nodes to a dummy root.

•The value of this dummy root is the combined frequency of its nodes and this root node is added back to the Min-heap.

•The procedure is then repeated until there is only one element left in the Min-heap.

<u>Steps to build Huffman Tree:</u>

1.The input to the algorithm is the array of characters in the text file.

2.The frequency of occurrences of each character in the file is calculated.

3.Struct array is created where each element includes the character along with their frequencies. They are stored in a priority queue (min-heap), where the elements are compared using their frequencies.

4.To build the Huffman tree, two elements with minimum frequency are extracted from the min-heap.

5.The two nodes are added to the tree as left and right children to a new root node which contains the frequency equal to the sum of two frequencies. A lower frequency character is added to the left child node and the higher frequency character into the right child node.

6.The root node is then again added back to the priority queue.

7.Repeat from step 4 until there is only one element left in the priority queue.

8.Finally, the tree’s left and right edges are numbered 0 and 1, respectively. For each leaf node, the entire tree is traversed, and the corresponding 1 and 0 are appended to their code until a leaf node is encountered.

9.Once we have the unique codes for each unique character in the text, we can replace the text characters with their codes. These codes will be stored in bit-by-bit form, which will take up less space than text.

Compressed File Structure:

We’ve talked about variable length input code generation and replacing it with the file’s original characters so far. However, this only serves to compress the file. The more difficult task is to decompress the file by decoding the binary codes to their original value.

This would necessitate the addition of some additional information to our compressed file in order to use it during the decoding process. As a result, we include the characters in our file, along with their corresponding codes. During the decoding process, this aids in the recreation of the Huffman tree.

Decompressing the Compressed File:

1. The compressed file is opened, and the number of unique characters and the total number of characters in the file are retrieved.
2. The characters and their binary codes are then read from the file. We can recreate the Huffman tree using this.
3. For each binary code:
   - A left edge is created for 0, and a right edge is created for 1.
   - Finally, a leaf node is formed and the character is stored within it.
   - This is repeated for all characters and binary codes. The Huffman tree is thus recreated in this manner.
4. The remaining file is now read bit by bit, and the corresponding 0/1 bit in the tree is traversed. The corresponding character is written into the decompressed file as soon as a leaf node is encountered in the tree.
5. Step 4 is repeated until the compressed file has been read completely.

In this manner, we recover all of the characters from our input file into a newly decompressed file with no data or quality loss.

Following the steps above, we can compress a text file and then overcome the bigger task of decompressing the file to its original content without any data loss.

Time Complexity: O(N * logN) where N is the number of unique characters as an efficient priority queue data structure takes O(logN) time per insertion, a complete binary tree with N leaves has (2*N – 1) nodes.

Techonology Used:

Implementation using C++ Language: