Discrete Mathematics Project

Data Compression using Huffman Code

Aaditya Makwana (202101112)
Aadey Shah (202101101)
Devang Rathod (202101204)
Vixita Bhalodiya (202101232)
Khushbu Maheshwari (202101512)

Index

1 Introduction

Discrete mathematics is the part of mathematics devoted to the study of discrete objects. It involves finite, rather than continuous mathematical structure. It involves integers, graph theory, algorithm and reasoning, matrix algebra, automata theory, cryptography, number theory, recursion and statements of logic. It has been describe as dealing with 'countable sets'. It excludes the continuous topics of mathematics or real numbers like: Euclidean mathematics and calculus.

Discrete mathematics has been describe as the mathematics of decision making for finite setting. It can be used to represent and solve problems with graphs. Modern computer science depends on discrete mathematics, especially graph theory and combinatorics.

So, the project which we have done gives us the opportunity to explore our little bit knowledge of discrete mathematics. In today's time people are heavily engrossed in a model technological world the amount of data that is being created and transferred is pretty huge. Images are important documents nowadays. It include various information. Compression of images is used in some applications such as profiling data and transmission system.

So, our project is based on image compression without losing any data. This project gives the application of 'Huffman coding and decoding algorithm' [1] which allows us to assign lesser bits to more frequently occurring data bits and vice versa.

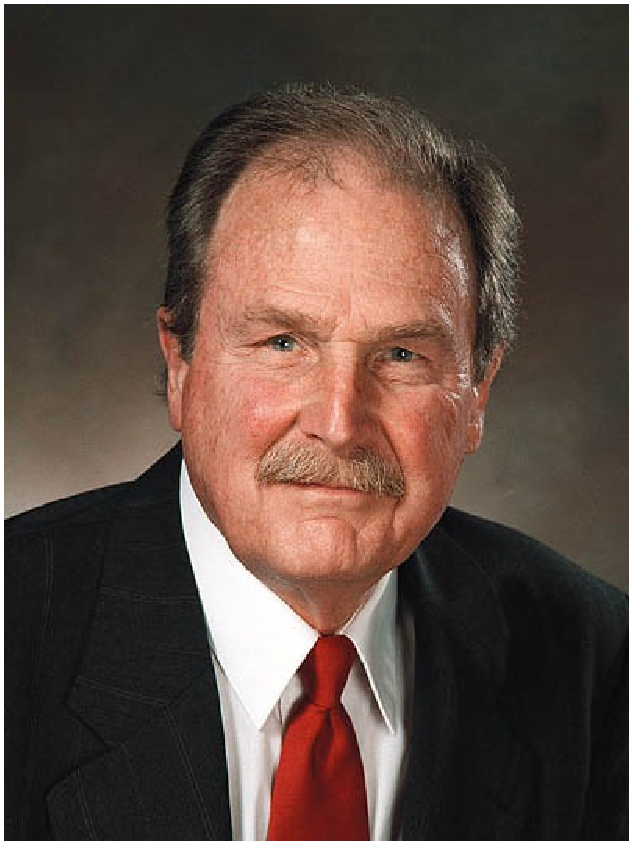
2 Abstract

This project makes an approach towards image compression using Huffman code. Image compression involves reducing the size of a given image without much loss of quality, and is carried out to reduce the amount of storage resources that an image requires. In the world full of data with shortage of resources, it is essential to make sure that the data is stored efficiently (by using data structures) and that the data is not unnecessarily large (by using data compression).

Image compression can be carried out using various methods, such as Discrete Cosine Transform, Fractal Compression, Colour Quantization and Huffman Code. However, we choose Huffman code because of its following advantages : It involves lossless compression of data and allows us to compress the data to a great extent. [2]

Huffman code was formulated by David Albert Huffman, an American pioneer in Computer Science, in 1952. This algorithm first appeared in his paper 'A Method for the Construction of Minimum-Redundancy Codes'.

Since then, it has appeared in many other works such as 'An Algorithm For Image Compression Using Huffman Coding Techniques' [3], by Sanjay Kumar Gupta, and 'Compression Using Huffman Coding' [4], by Mamta Sharma.


David A. Huffman (August 9, 1925 – October 7,1999)

3 Mathematics - Huffman Code

There are three ways to measure the efficiency of Huffman code as follows: -

1. Bitrate
2. Entropy
3. Compression Ratio

The above methods are discussed in brief below: -
Bitrate: Bitrate (R) gives us the efficiency of all the various cases of the binary

tree. It is defined as the average number of expected bits to represent a symbol. It can be calculated as follows: -

$$\mathrm{R} = \sum_{\mathbf{l=1}}^{\mathbf{L}} \mathbf{p_l n_l}$$

Here pl denotes the probability of the symbol and nl denotes the number of bits in the codeword assigned to the symbol.
Entropy: Denoted by H(X), is given by: -

$$\mathrm{H(X)} = \sum_{\mathbf{i=1}}^{\mathbf{N}} -\mathbf{p_i} * \mathbf{log_2 p_i}$$

It gives us the idea about how much information is stored in a bit.
Compression Ratio: It is the ratio of the difference of size of original data and compressed data to size of original data. In other words, it gives us the idea about the number of bits freed/saved per bit in the compressed data.

Compression Ratio = (Original Size - New Size)/(Original Size)

Decoding:
The compression of data can be carried out after the codewords have been assigned to each symbol. However, for decoding, the compressed file must contain the table with the original symbols and the codewords associated to it. Upon using the table and codewords, the compressed data can be extracted.

4 Algorithm

Now we take an example of a 7x7 sample image having size 4 bits to understand the working of the algorithm. It consists of 8 steps starting from calculating the probability of an element to occur to generating a key table so as to be able to decode the data.

Step 1 – For calculating probability of each symbol we have to first store each symbol in a list and then compute the frequency. For storing each symbol in a list, we have to push element whenever we encounter a symbol while iterating whole matrix given. Subsequently for frequency, we can initially set the frequency of that symbol as zero and then increase the frequency of that symbol by one whenever we encounter that symbol. After calculating the frequency of each symbol, we can calculate the probability by the following formula:

$\Rightarrow \mathrm{P}_{\mathbf{symbol}} = \mathrm{Freq}_{\mathbf{symbol}}$ / $\mathrm{Total}_{\mathbf{symbol}}$
$\rightarrow \mathrm{P}_{\mathbf{symbol}}$ = Probability of that symbol
$\rightarrow \mathrm{Freq}_{\mathbf{symbol}}$ = Frequency of that symbol
$\rightarrow \mathrm{Tota}_{\mathbf{symbol}}$ = Total number of symbols

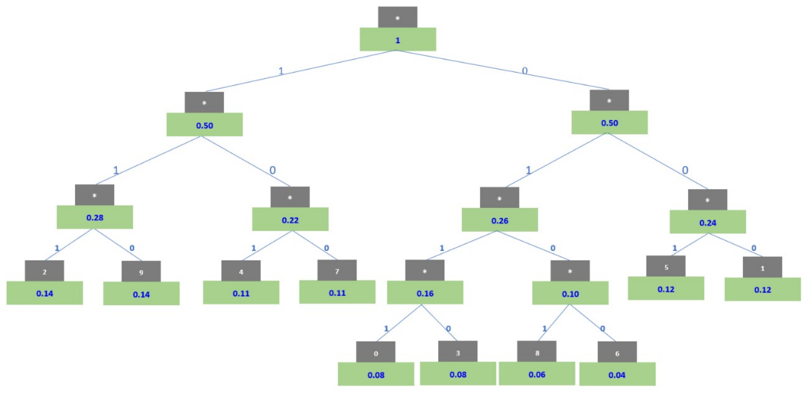
Step 2 – Now after calculating the probability, we have to create a table with symbol, frequency and probability in order to create binary tree. Table should be created in such a way that it is in descending order. The table for corresponding data will be:

7	8	0	2	4	6	5
5	2	3	9	7	1	0
1	9	2	7	0	3	4
4	1	2	8	9	7	5
3	1	4	6	3	9	1
2	9	7	9	5	2	1
5	0	9	2	4	5	8

Step 3 – For creating the probability tree such that the most frequent appearing symbol is assigned the small codeword which is indeed called greedy method of compression. Now for that we have to create a pseudo node containing the address of the left and right child, probability and symbol. Each pseudo node consisting left and the right child as the current minimum probabilities, and parent node as the sum of its child node. Now link this pseudo node with another pseudo node until the probability of root of the tree becomes one.

Symbol	Frequency	Probability
2	7	0.14
9	7	0.14
5	6	0.12
1	6	0.12
4	5	0.11
7	5	0.11
0	4	0.08
3	4	0.08
8	3	0.06
6	2	0.04

Step 4 – We now start assigning code words to the symbols present in the binary tree. For this, we first search the symbol in the binary tree, and traverse the tree starting from the root and ending at the leaf node where the respective symbol is stored. During the process of traversing, if we traverse through a left branch, we append a "1" to the codeword string, and if we traverse through a right branch, we append a "0" to the codeword string. We repeat this until the leaf node is reached. The final string thus obtained is the codeword for the particular symbol stored in the respective leaf node. We repeat this process for all the symbols and store the results.



Step 5 – We are finally done with the process, and after following the above steps, we will be able to encode and decode a particular form of data. If the user wishes to calculate that factor with which the data has been compressed, we may calculate the compression percentage, which is given by :

⇒ Percentage = ((Original Size – New Size)/(Original Size))*100

5 Implementation using OOP's

```
#include <iostream>
#include <string>
#include <queue>
#include <unordered_map>
using namespace std;
#define EMPTY_STRING " "

struct Node
{
    char ch;
    int frequency;
    Node *left, *right;
};

Node *getNode(char ch, int frequency, Node *left,
              Node *right)
{
    Node *node = new Node();
    node->ch = ch;
    node->frequency = frequency;
    node->left = left;
    node->right = right;
    return node;
```

```
}

struct compare
{
    bool operator()(const Node *left, const Node *right)
        const
    {
        // the highest priority item has the lowest frequency
        return left->frequency > right->frequency;
    }
};

// to check if Huffman Tree contains only a single node
bool validLeaf(Node *root)
{
    return root->left == nullptr && root->right == nullptr;
}

void Encode(Node *root, string str, unordered_map<char, string> &huffmanCode)
{
    if (root == nullptr)
    {
        return;
    }
    if (validLeaf(root))
    {
        huffmanCode[root->ch] = (str != EMPTY_STRING) ? str : "1";
    }
    Encode(root->left, str + "0", huffmanCode);
    Encode(root->right, str + "1", huffmanCode);
}

void Decode(Node *root, int &index, string str)
{
    if (root == nullptr)
    {
        return;
    }
    if (validLeaf(root))
    {
        cout << root->ch;
        return;
    }

    index++;
    if (str[index] == '0')
    {
        Decode(root->left, index, str);
    }
    else
    {
        Decode(root->right, index, str);
    }
}

void HuffmanTree(string symbol)
{
    if (symbol == EMPTY_STRING)
    {
        return;
    }

    unordered_map<char, int> frequency;
    for (char ch : symbol)
    {
        frequency[ch]++;
    }

    priority_queue<Node *, vector<Node *>, compare> pq;
    for (auto pair : frequency)
    {
        pq.push(getNode(pair.first, pair.second, nullptr,
                        nullptr));
    }

    while (pq.size() != 1)
    {
        Node *left = pq.top();
        pq.pop();
        Node *right = pq.top();
        pq.pop();
        int sum = left->frequency + right->frequency;
        pq.push(getNode('\0', sum, left, right));
    }

    Node *root = pq.top();
    unordered_map<char, string> huffmanCode;
    Encode(root, EMPTY_STRING, huffmanCode);

    cout << "Huffman Codes a r e : \n"
         << endl;
    cout << "== == == == == == == == == == ==" << endl;
    for (auto pair : huffmanCode)
    {
        cout << pair.first << " --> " << pair.second << endl;
    }

    cout << "== == == == == == == == == == ==" << endl;
    cout << "\nThe original string is : \n"
         << symbol << endl;

    // Print encoded string
    string str;
    for (char ch : symbol)
    {
        str += huffmanCode[ch];
    }
    cout << "\nThe encoded string i s : \n"
         << str << endl;
    cout << "\nThe decoded string i s : \n";

    if (validLeaf(root))
    {
        // Special case : For input like a , aa , aaa , etc .
        while (root->frequency--)
        {
            cout << root->ch;
        }
    }
    else
    {
        int index = -1;
        while (index < (int)str.size() - 1)
        {
            Decode(root, index, str);
        }
    }
}

int main()
{
    string symbol = " Test runs are mandatory to ensure that the code works.";

    HuffmanTree(symbol);
    return 0;
}
```

6 Commercialization

• Since data is used in everyday life, data compression is also a necessary tool that needs to be implemented for efficient use of resources. Hence, this algorithm can be implemented in software/websites that allow users to compress a given image, or any data for that matter.

• Huffman coding is also used in transmission of fax and text messages.

• We may also use some other algorithms along with Huffman coding in our software/website and add options to compress data types like pdf, text, etc.

References

[1] https://en.m.wikipedia.org/wiki/Huffman_coding.

[2] https://stackoverflow.com/questions/21853101/why-huffman-coding-is-good.

[3] Sanjay Kumar Gupta. An Algorithm For Image Compression Using Huffman Coding Techniques. *International Journal of Advance Research in Science and Engineering*, 5, 2016.

[4] Mamta Sharma. Compression Using Huffman Coding. *IJCSNS International Journal of Computer Science and Network Security*, 10(5), 2010.