DATA COMPRESSION

BY LZW AND HUFFMAN ALGORITHMS

CSE2003: Data Structures and Algorithm
Submitted to: Govinda K

TANYA GUPTA(20BEC0740)

SUBHASREE.V(20BEC0552)

AIM

The need for speed and storage space is the focus of the latest computer technology. Multimedia files are large and consume hard disk space. File size makes it time consuming to move from one place to another through school networks or stream them online. Squeezing shrinks files, making them smaller and more useful for storing and sharing. The aim of our project is to compress short text in order to reduce memory usages. Data compression is a technique which is used to reduce the file size by encoding, restructuring, and decoding. The aim of the project is to compare two different data compression algorithms namely, LZW algorithm and Huffman algorithm.

ABSTRACT

Data compression is widely used In GIFS, PDF, TIFF etc. Data compression is the process by which different files are used such as text, audio, and video can be converted to another file, so that the original file can be fully accessed original file without loss of real information. This process is important if it is needed to save storage space.

Data compression has many benefits. Computer hardware storage, data transfer time, and network bandwidth are all considered when we talk about the benefits of data compression. There are two types of data compression which are lost and non-lost. Missing data compression does not release any data and can be undone at any time. On the other hand, the lost data compress releases unwanted data and cannot be reverted back to the original data. LZW is an easy-to-use data compression method. It works by reading a set of symbols, and converting them into codes. Whenever a new character is introduced, the sequence is coded and added as a message. So this helps to compress the message as the sequence is different from the given codes. Huffman encoding, on the other hand, is also a way of compressing lost data when flexible code codes are assigned to characters whose length is based on corresponding character frequencies. The smaller code is given the most common letter, while the larger code is given the lower case letter. This project will be done in C ++ to successfully complete all the standard features of the project.

Huffman Coding is a mathematical data modifier that provides the reduction of the code length used to represent letters of the alphabet. This is a common form of code that does not require minimal code. LZW dictionary the most popular based pressure tool. This means that instead of spraying character numbers and building trees LZW encrypts the data by reference to the dictionary. Compared to any flexible and flexible pressure method, the concept is to start with the first model, read the data and update the draft and code of the data. In this paper, we investigate. Next question: Which coding, LZW or Huffman, is more appropriate compared to the other? LZW does not require prior information about input data transmission, and LZW may press input one world that allows for immediate execution. LZW encoding is most likely when the pressure level is high and low decompression time is required.

INTRODUCTION

The term Algorithm means “the process or set of rules that must be followed in calculating or other problem-solving activities”. The algorithm therefore refers to a set of rules / guidelines that explain step by step how the work should be done in order to obtain the expected results. The algorithm is a clever step to represent a solution to a particular problem. In Algorithm the problem is broken down into smaller pieces or steps so it is easier for the editor to convert it into a real program.

Lempel – Ziv – Welch (LZW) is a global data loss algorithm developed by Abraham Lempel, Jacob Ziv, and Terry Welch. Published by Welch in 1984 as an advanced implementation of the LZ78 algorithm published by Lempel and Ziv in 1978. The algorithm is easy to use and has very high performance in hardware usage. It is a widely used Unix file compression algorithm and in GIF image format. LZW compression works by learning the sequence of symbols, collecting symbols into strings, and converting strings into codes. Because the codes take up less space than the cables they replace, we get compressed. The concept of the compression algorithm is as follows: as the input data is still processed, the dictionary maintains a connection between the long words encountered and the list of code values. Names are replaced by their corresponding codes so the input file is compressed. Therefore, the efficiency of the algorithm increases as the number of long, repetitive words in the input data increases.

Huffman Coding is a way of compressing data to reduce its size without losing any data. It was first founded by David Huffman. Huffman Coding is often useful for compressing data when characters are common.

Huffman coding is an algorithm for compressing lost data. The idea is to provide flexible length codes to input characters, the length of the given codes is based on the corresponding character frequencies. The most common character gets the least code and the most common character gets the biggest code. Variable length codes are given input Initial Codes, meaning that the codes (sequence of bits) are assigned in such a way that the code assigned to one character is not the beginning of the code assigned to any other character. This is how Huffman Coding ensures that there is no ambiguity when coding the generated bitstream code.

LITERATURE REVIEW

1. **Analysis of Huffman Coding and Lempel–Ziv–Welch (LZW) Coding as Data Compression Techniques**
   by Gajendra Sharma Kathmandu University, School of Engineering, Department of Computer Science and Engineering, Dhulikhel, Kavre, Nepal Author's Mail id: gajendra.sharma@ku.edu.np

   In this research paper, they have focused on these algorithms to check differences such as compression time, decompression time and compression ratio. Huffman coding algorithm is well-situated than LZW coding. LZW coding has more compression ratio than Huffman algorithm. While Huffman coding requires more execution time than the LZW. In cases where time is not important Huffman coding can be used to obtain high compression ratio. Whereas for some applications the time is important for real-time applications and hence LZW coding.

2. **Performance comparison of Huffman and Lempel-Ziv Welch data compression for wireless sensor node application**
   By Asral Bahari Jambek and Nor Alina Khairi School of Microelectronic Engineering, Universiti Malaysia Perlis, Pauh Putra Campus, Perlis, Malaysia

   This study analyzes the effective efficacy of Huffman algorithm and LZW algorithm using various input data usually measured wireless sensor node, i.e. temperature, humidity, ECG and text data. With the tested data provided, Huffman. The algorithm shows better performance compared in LZW at a pressing rate once calculation time. From testing, Huffman The algorithm is able to achieve a data rate of 43% reduction. With double pressure, LZH can provide up to 9% improvement depending on the data reduction, but at an increased cost of calculation time. In the future, this work will continue learn various techniques for representing WSN data to further increase the efficiency of the Huffman algorithm.

3. PERFORMANCE COMPARISON OF HUFFMAN CODING AND LEMPEL-ZIV-WELCH TEXT COMPRESSION ALGORITHMS WITH CHINESE REMAINDER THEOREM

By Mohammed Babatunde IBRAHIM, Kazeem Alagbe GBOLAGADE
Department of Computer Science,Kwara State University, Malete, Nigeria

The study provided performance comparisons between the two texts is the compression algorithm while incorporating Chinese Remainder Theorem (CRT) for them so you have to see what it is done better. By compression size, Huffman-CRT did a little better than that LZW-CRT at 55.64Kb to 57.11Kb, for time to press Huffman-CRT again performed better than LZW-CRT with 44.61s to 55.39s equally in terms of pressure, Huffman-CRT also did better than that LZW-CRT 3.70 times to 2.37 times. Therefore, the results reveal hopes to develop documentary presentations through CRT and other file formats. In future jobs, there are the need to supply other compression algorithms such as Run Length Encoding, Discrete wave Conversion, Adaptive Huffman Coding and Arithmetic Code Text with CRT.

PROPOSED METHOD

1. LZW ALGORITHM

- Every character is typically stored with 8 binary bits, giving the data up to 256 distinct symbols.
- This technique aims to increase the library's character size from 9 to 12 bits. The new one-of-a-kind symbols are built up of symbol combinations that previously appeared in the string.
- It doesn't always compress properly, especially when the strings are short and varied. However, it is useful for compressing redundant data and does not require saving a new dictionary with the data because this method compresses and decompress data.

EXAMPLE:

I WANT TO COMPRESS THE FOLLOWING STRING: thisisthe

Start adding pair of symbols to dictionary of words. When we reach a pair that we've already placed in the dictionary. Replace that pair with our new dictionary value.

Current	Next	Output	Add to dictionary
t 116	h 104	t 116	th 256
h 104	i 105	h 104	hi 257
i 105	s 115	i 105	is 258
s 115	i 105	s 115	si 259
i 105	s 115	"is" is in dictionary!	Check if "ist" is
is 258	t 116	is 258	ist 260
t 116	h 104	"th" is in the dictionary!	Check if "the" is.
th 256	e 101	th 256	the 261
e 101	-	e 101	-

Output

116 104 105 115 258 256 101

I WANT TO DECOMPRESS THE FOLLOWING STRING:

116 104 105 115 258 256 101

Current	Next	Output	Add to dictionary
116	104	116	116 104 [256]
104	105	104	104 105 [257]
105	115	105	105 115 [258]
115	258	115	115 105 115 [259]
258	256	105 115	105 115 116 [260]
256	101	116 104	116 104 101 [261]
101	-	101	-

Output
thisisthe

PSEUDOCODE:

ENCODING:

```
1   Start table with single character string
2   S = first input character
3   WHILE till the end of input stream
4       K = next input character
5       IF S + K is already present in the string table
6         S = S + K
7       ELSE
8       output the code for S
9       add S + K to the string table
10        S = K
11      END WHILE
12   output code for S
```

DECODING:

1 Start table with single character string
2 old = first input code
3 output translation of old
4 WHILE not end of input stream
5 new = next input code
6 IF new is not present already in the string table
7 P = translation of old
8 P = P + K
9 ELSE
10 P = translation of new
11 output P
12 K = first character of P
13 old + K to the string table
14 old = new
15 END WHILE

CODE:

```
#include <bits/stdc++.h>
#include <unordered_map>
using namespace std;
vector<int> encoding(string s1)
{
    cout << "Encoding:\n";
    unordered_map <string , int> table;
    for (int i = 0; i <= 255; i++) {
        string ch = "";
        ch += char(i);
        table[ch] = i;
    }

    string p = "", c = "";
    p += s1[0];
    int code = 256;
    vector<int> output_code;
    cout << "String\tOutput_Code\tAddition\n";
    for (int i = 0; i < s1.length(); i++) {
        if (i != s1.length() - 1)
            c += s1[i + 1];
        if (table.find(p + c) != table.end()) {
            p = p + c;
        }
        else {
            cout << p << "\t" << table[p] << "\t\t"
                 << p + c << "\t" << code << endl;
            output_code.push_back(table[p]);
            table[p + c] = code;
            code++;
```

```
            p = c;
        }
        c = "";
    }
    cout << p << "\t" << table[p] << endl;
    output_code.push_back(table[p]);
    return output_code;
}

void decoding(vector<int> op)
{
    cout << "\nDecoding:\n";
    unordered_map<int, string> table;
    for (int i = 0; i <= 255; i++) {
        string ch = "";
        ch += char(i);
        table[i] = ch;
    }
    int old = op[0], n;
    string s = table[old];
    string c = "";
    c += s[0];
    cout << s;
    int count = 256;
    for (int i = 0; i < op.size() - 1; i++) {
        n = op[i + 1];
        if (table.find(n) == table.end()) {
            s = table[old];
            s = s + c;
        }
```

```
        else {
            s = table[n];
        }
        cout << s;
        c = "";
        c += s[0];
        table[count] = table[old] + c;
        count++;
        old = n;
    }
}
int main()
{
    string s;
    cout<<"Enter the string to be compressed: ";
    cin>>s;
    vector<int> output_code = encoding(s);
    cout << "Output Codes are: ";
    for (int i = 0; i < output_code.size(); i++) {
        cout << output_code[i] << " ";
    }
    cout << endl;
    decoding(output_code);
}
```

OUTPUT:

Output Clear

```
/tmp/jysteRhXKd.o
Enter the string to be compressed:
Data#Structures
Encoding:
String  Output_Code Addition
D   68      Da  256
a   97      at  257
t   116     ta  258
a   97      a#  259
#   35      #S  260
S   83      St  261
t   116     tr  262
r   114     ru  263
u   117     uc  264
c   99      ct  265
t   116     tu  266
u   117     ur  267
r   114     re  268
e   101     es  269
```

Output Clear

```
a   97      at  257
t   116     ta  258
a   97      a#  259
#   35      #S  260
S   83      St  261
t   116     tr  262
r   114     ru  263
u   117     uc  264
c   99      ct  265
t   116     tu  266
u   117     ur  267
r   114     re  268
e   101     es  269
s   115
Output Codes are: 68 97 116 97 35 83 116 114 117 99 116 117 114 101
    115

Decoding:
Data#Structures
```

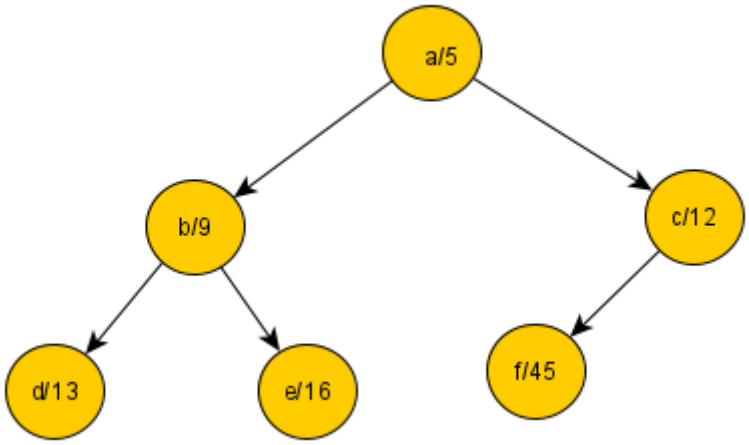
2. HUFFMAN CODING ALGORITHM

.

APPROACH:

Character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

1)The first step is to build a min heap for each unique character. This heap would be based on frequencies of the input characters such that each node data is less than (or equal to) the data in that node's children.



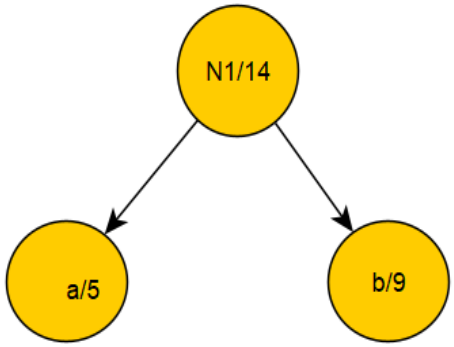
Min Heap

2) Once the min heap is created, two nodes with minimum frequencies are to be extracted. This step is greedy in nature.

3) In the next step, a new internal node with frequency equal to the sum of the two nodes frequencies is formed. The first extracted node forms its left child and the second extracted node becomes right child. After node formation, this new internal node is added to our min heap.

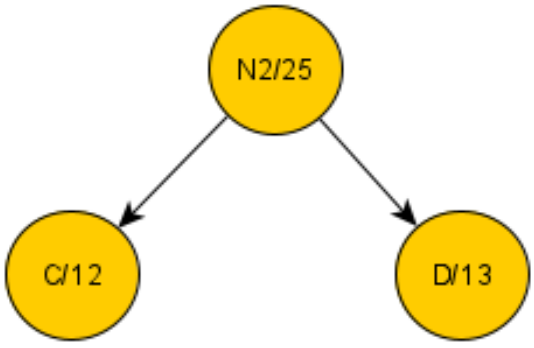
4)The last step is to perform step 2&3 until only a single node remains in our heap.

(i) Here we have character ‘a’ and ‘b’ with the least frequencies.



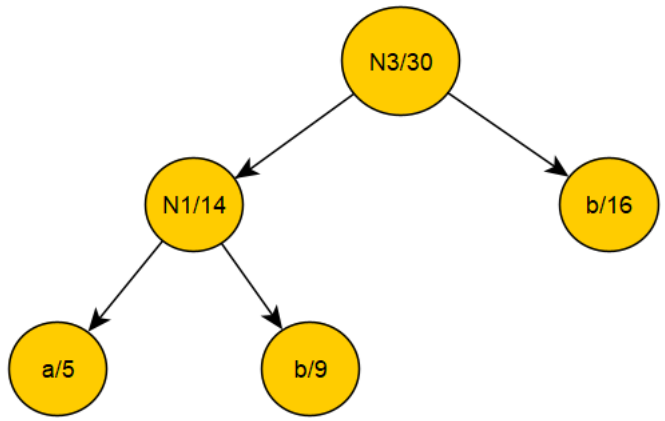
Character	Frequency
N1	14
c	12
d	13
e	16
f	45

(ii) So we again extract the two nodes with minimum frequencies. Here we have ‘c’ and ‘d’ with least frequencies. We form a new internal node and add node ‘c’ to its left and node ‘d’ to its right and its frequency as the sum of its children. And finally add this new internal node to our mean heap.



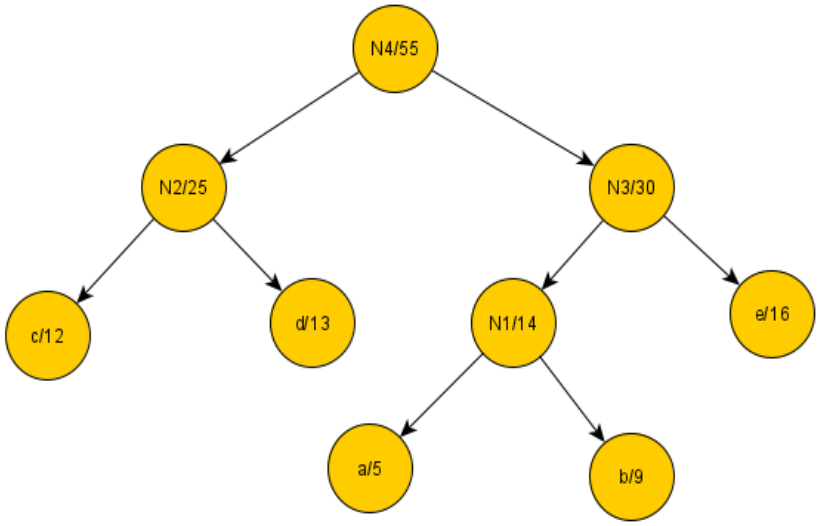
Character	Frequency
N1	14
N2	25
e	16
f	45

(iii) Again we extract the two nodes with minimum frequencies. Here we have ‘N1’ and ‘e’ with least frequencies. We form a new internal node and add node ‘N1’ to its left and node ‘e’ to its right and its frequency as the sum of its children.



Character	Frequency
N2	25
N3	30
f	45

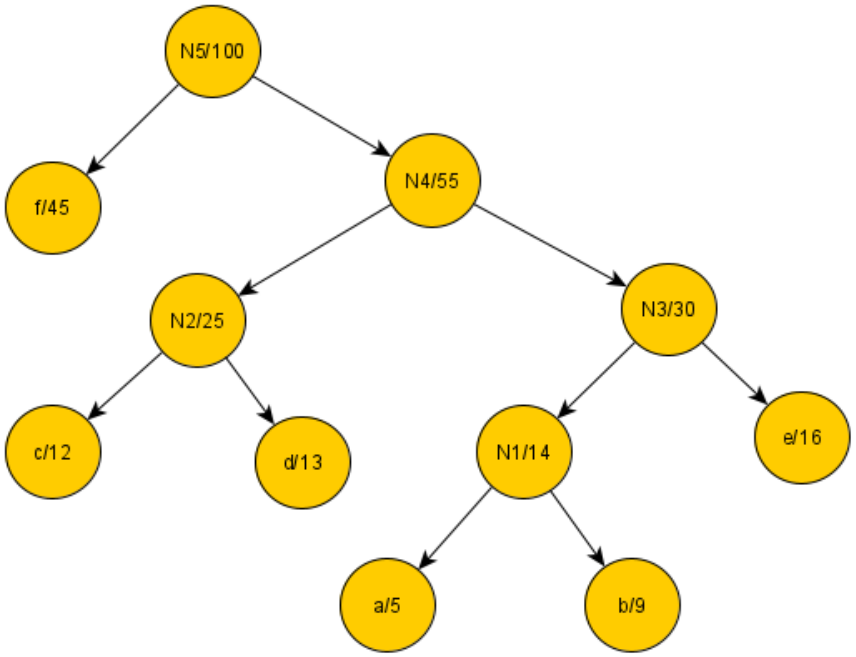
(iv) So again we extract the two nodes with minimum frequencies. Here we have ‘N2’ and ‘N3’ with least frequencies. We form a new internal node and add node ‘N2’ to its left and node ‘N3’ to its right and its frequency as the sum of its children.



Character	Frequency
N4	55
f	45

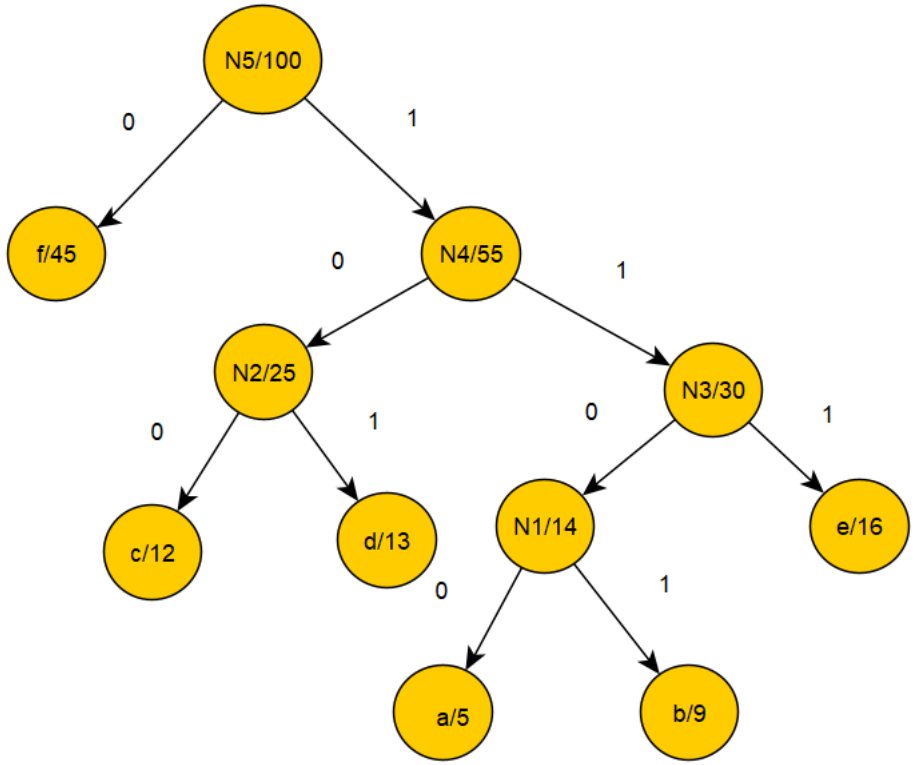
(v) And finally add this new internal node ‘N4’ to our min-heap. We now have 2 nodes left, merging them would give a single node and thus our algorithm may stop. So we form a new internal node ‘N5’ and add ‘f’ with lesser frequency to its left and ‘N4’ with higher frequency to its right and its frequency as the sum of its two children.

(vi)So this would be the final output of this algorithm and we'll call it a Huffman tree.



Huffman Tree

(vii)Now to generate code from this tree, we'll traverse it from the root and assign 0 to the left child and 1 to the right child. Once we encounter a leaf node, we may print this code.



This would be the codes for each character.

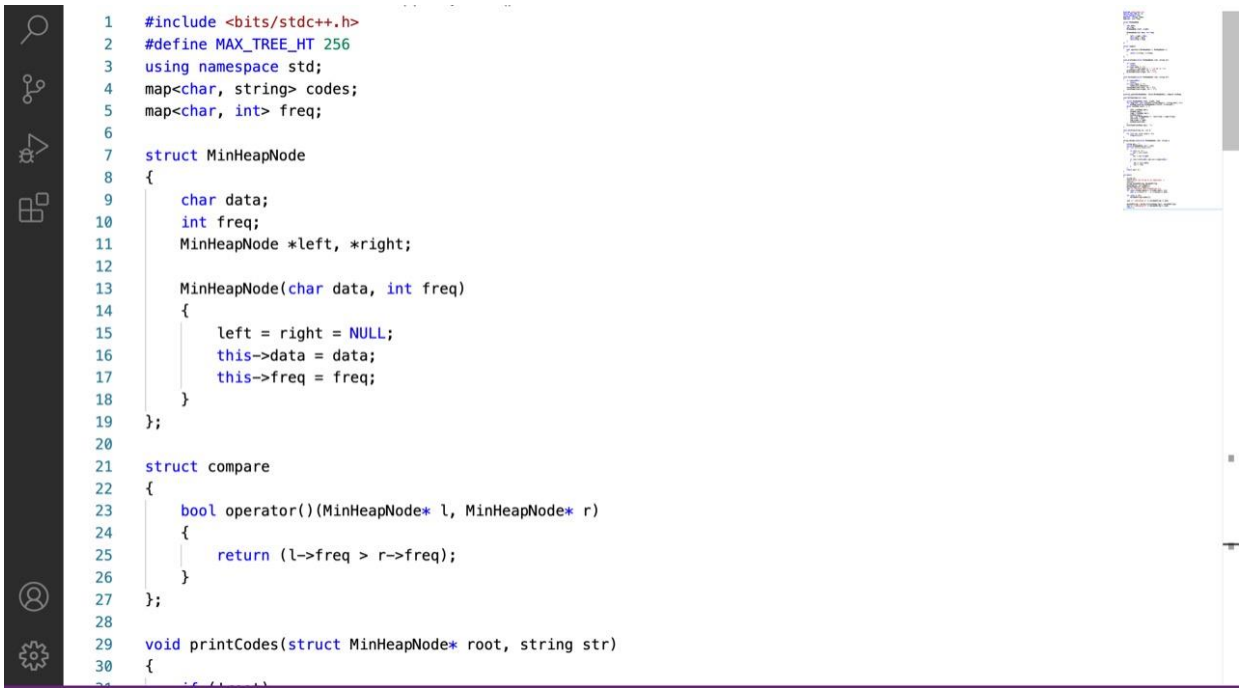
Character	Code
a	1100
b	1101
c	100
d	101
e	111
f	0

PSEUDOCODE:

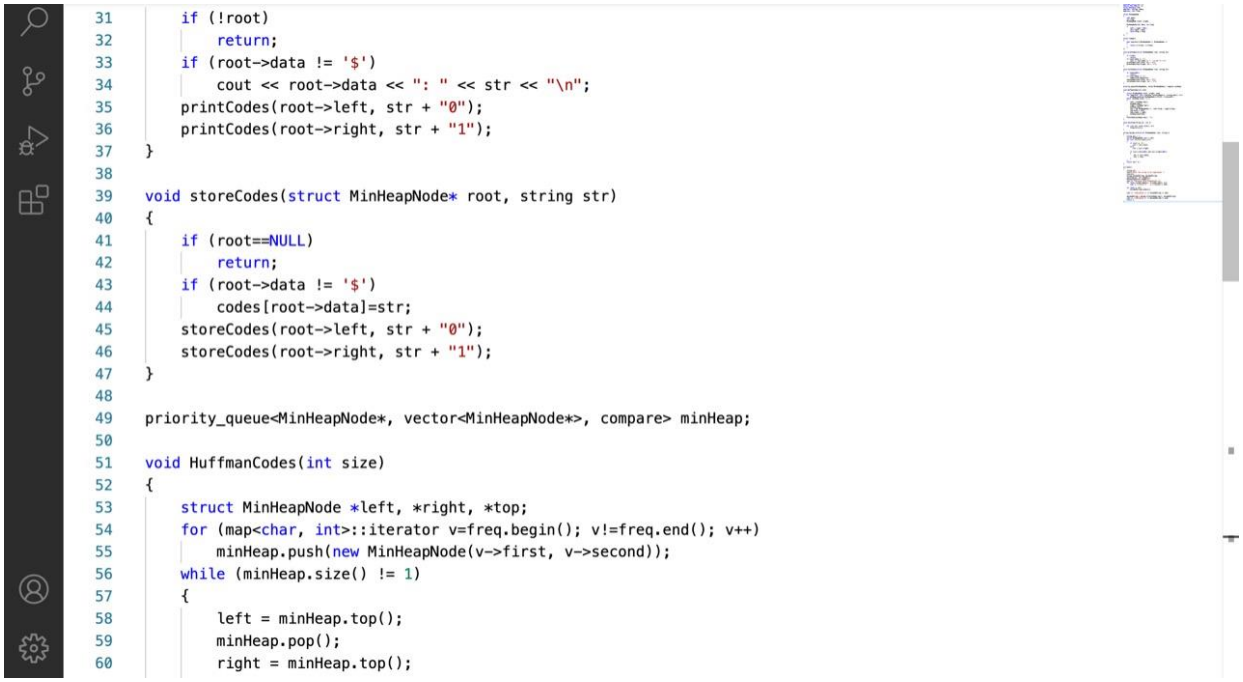
1. Each character is matched to its huffman value and the frequency of character of the input data is stored.
2. A Huffman tree node is created by assigning one of the input characters and the frequency of the character and its left and right child.
3. Utility functions are introduced for the priority queue, to print characters along with their huffman value
4. STL priority queue to store heap tree, with respect to their heap root node value is also introduced.
5. Function to build the Huffman tree and store it in minheap is created
6. This function iterates through the encoded string s where,

- if s[i]=='1' then move to node->right
- if s[i]=='0' then move to node->left
- if leaf node append the node->data to our output string
- reached leaf node
- cout<<ans<<endl;

CODE:



```cpp
#include <bits/stdc++.h>
#define MAX_TREE_HT 256
using namespace std;
map<char, string> codes;
map<char, int> freq;

struct MinHeapNode
{
    char data;
    int freq;
    MinHeapNode *left, *right;

    MinHeapNode(char data, int freq)
    {
        left = right = NULL;
        this->data = data;
        this->freq = freq;
    }
};

struct compare
{
    bool operator()(MinHeapNode* l, MinHeapNode* r)
    {
        return (l->freq > r->freq);
    }
};

void printCodes(struct MinHeapNode* root, string str)
{
```



```cpp
    if (!root)
        return;
    if (root->data != '$')
        cout << root->data << ": " << str << "\n";
    printCodes(root->left, str + "0");
    printCodes(root->right, str + "1");
}

void storeCodes(struct MinHeapNode* root, string str)
{
    if (root==NULL)
        return;
    if (root->data != '$')
        codes[root->data]=str;
    storeCodes(root->left, str + "0");
    storeCodes(root->right, str + "1");
}

priority_queue<MinHeapNode*, vector<MinHeapNode*>, compare> minHeap;

void HuffmanCodes(int size)
{
    struct MinHeapNode *left, *right, *top;
    for (map<char, int>::iterator v=freq.begin(); v!=freq.end(); v++)
        minHeap.push(new MinHeapNode(v->first, v->second));
    while (minHeap.size() != 1)
    {
        left = minHeap.top();
        minHeap.pop();
        right = minHeap.top();
```

```
        minHeap.pop();
        top = new MinHeapNode('$', left->freq + right->freq);
        top->left = left;
        top->right = right;
        minHeap.push(top);
    }
    storeCodes(minHeap.top(), "");
}

void calcFreq(string str, int n)
{
    for (int i=0; i<str.size(); i++)
        freq[str[i]]++;
}

string decode_file(struct MinHeapNode* root, string s)
{
    string ans = "";
    struct MinHeapNode* curr = root;
    for (int i=0;i<s.size();i++)
    {
        if (s[i] == '0')
           curr = curr->left;
        else
           curr = curr->right;

        if (curr->left==NULL and curr->right==NULL)
        {
            ans += curr->data;
            curr = root;
        }
    }
    return ans+'\0';
}

int main()
{
    string str;
    cout<<"Enter the string to be compressed: ";
    cin>>str;
    string encodedString, decodedString;
    calcFreq(str, str.length());
    HuffmanCodes(str.length());
    cout << "Character With Frequencies:\n";
    for (auto v=codes.begin(); v!=codes.end(); v++)
        cout << v->first <<' ' << v->second << endl;

    for (auto i: str)
        encodedString+=codes[i];

    cout << "\nEncoding:\n" << encodedString << endl;

    decodedString = decode_file(minHeap.top(), encodedString);
    cout << "\nDecoding:\n" << decodedString << endl;
    return 0;
}
```

OUTPUT:

Output Clear

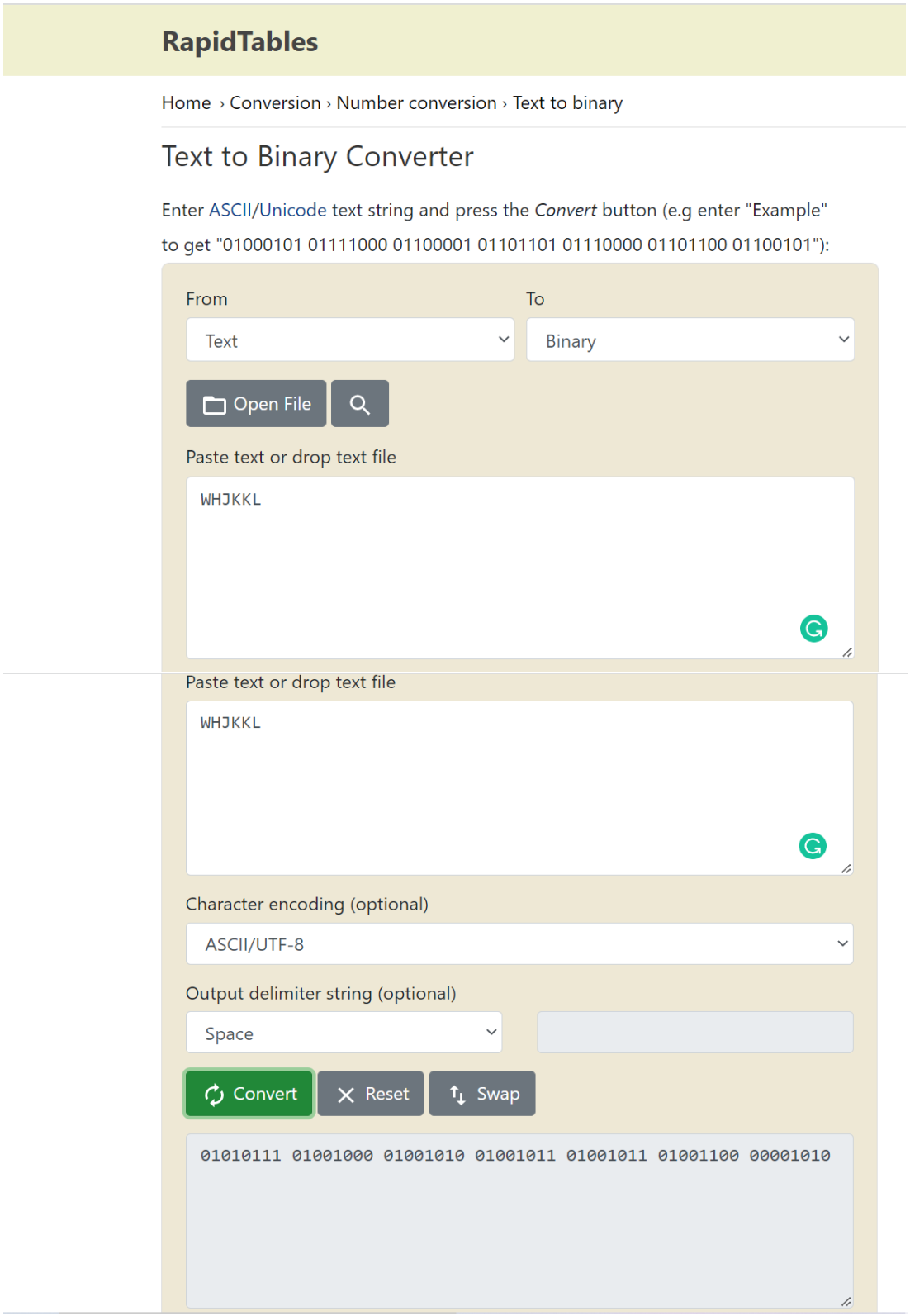
```
/tmp/pWOAAJR8Q3.o
Enter the string to be compressed:
Data#Structures
Character With Frequencies:
# 0110
D 1101
S 0111
a 111
c 0100
e 1100
r 100
s 0101
t 00
u 101

Encoding:
1101111001110110011100100101010000101100110001 01

Decoding:
```

Output Clear

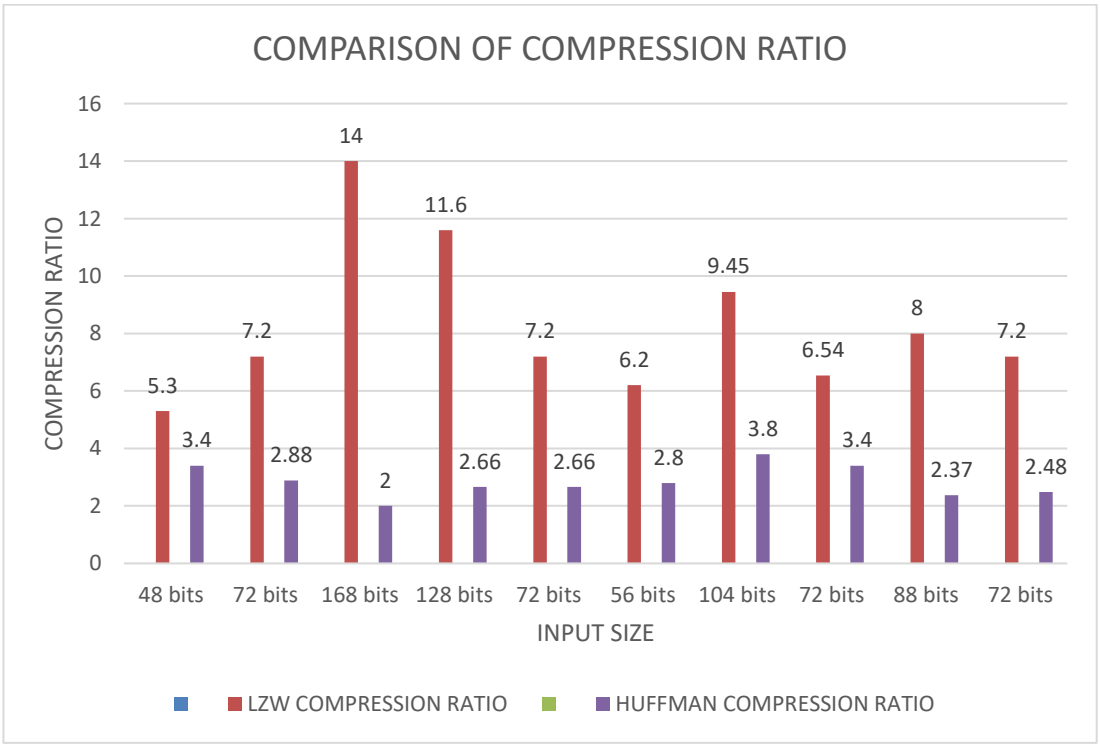
```
Data#Structures
Character With Frequencies:
# 0110
D 1101
S 0111
a 111
c 0100
e 1100
r 100
s 0101
t 00
u 101

Encoding:
1101111001110110011100100101010000101100110001 01

Decoding:
Data#Structures
```

COMPARISON

For finding the size of the text and ASCII in bits, the following application is used.



S.NO	TEXT	INPUT SIZE	LZW COMPRESSED SIZE	LZW COMPRESSION RATIO	HUFFMAN COMPRESSED SIZE	HUFFMAN COMPRESSION RATIO
1.	WHJKKL	48 bits	9 bits	5.3	14 bits	3.4
2.	Huffman**	72 bits	10 bits	7.2	25 bits	2.88
3.	Comparisonlzw&huffman	168 bits	12 bits	14	84 bits	2
4.	Data#Structures	128 bits	11 bits	11.6	48 bits	2.66
5.	ujhfghksc	72 bits	10 bits	7.2	27 bits	2.66
6.	@#%^&*(	56 bits	9 bits	6.2	20 bits	2.8
7.	ELEVEN#ELEVEN	104 bits	11 bits	9.45	27 bits	3.8
8.	thisisthe	72 bits	11 bits	6.54	21 bits	3.4
9.	python&java	88 bits	11 bits	8	37 bits	2.37
10.	Something	72 bits	10 bits	7.2	29 bits	2.48



COMPRESSION RATIO:

In this, we've compared the compression ratios of LZW algorithm and Huffman algorithm. Compression ratio is that the ratio of the uncompressed size to the compressed size. This ratio are often helpful in indicating complexity of data set or signal. It's helpful in seeing what quantity of a file is compressed without increasing its actual size.

Compression ratio=Uncompressed size/Compressed size

As we observe the graph above, LZW algorithm has high compression ratios for texts of small input size, while Huffman algorithm has low compression ratios for the same. When the compression ratios are high, it implies that for the actual input text size, the used algorithm is acceptable and might yield good compression results. Therefore, we will conclude that LZW algorithm works better for texts of small input size and works proficiently in terms of small text compression.

TIME COMPLEXITY:

Time complexity of an algorithm will be a good indicator of the performance of a algorithm. As the dictionary size is fixed and independent of the input length, LZW yields a time complexity of O(n) as each byte is barely read once and also the complexity of the operation for every character is constant. The time complexity of the Huffman algorithm is O(nlogn). Employing a heap to store the weight of every tree, each iteration requires O(logn) time to work out the cheapest weight and insert the new weight. There are O(n) iterations, one for every item. This suggests that LZW algorithm also works better in terms of time complexity because it is quicker.

CONCLUSION

ADVANTAGES OF LZW ALGORITHM OVER HUFFMAN ALGORITHM:

- No prior information about the input data stream is given to the LZW algorithm.
- Input stream can be compressed in one single pass by LZW algorithm.
- LZW algorithm is simple and easy to use, allowing for a fast execution.
- LZW algorithm has a high compression ratio for small size texts which proves as a big advantage.
- LZW is also a faster working algorithm.

DISADVANTAGES OF LZW ALGORITHM:

- Management of the string table can be complex.
- Files without repetitive data cannot be compressed much.
- LZW algorithm works well for text files but not so much other type of files based on past researches.
- The amount of storage required can be undeterminable.

ADVANTAGES OF HUFFMAN ALGORITHM:

- Huffman algorithm mostly has high compression ratios for some types of files.
- The decoding can be efficient.
- It is uniquely decodable.

DISADVANTAGES OF HUFFMAN ALGORITHM:

- Each symbol is coded separately.
- Each symbol uses whole number of bits.
- It can be very inefficient when assigned with unlikely values.
- Encoding can be slow as two passes are needed at the input.

REFERENCES

https://www.isroset.org/pub_paper/IJSRCSE/5-ISROSET-IJSRCSE-02950.pdf

https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.947.1827&rep=rep1&type=pdf

http://bulletin.feccupit.ro/archive/pdf/2019_2_2.pdf

https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

https://en.wikipedia.org/wiki/Huffman_coding

https://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique/