Mini Project on

"Comparative Study of Huffman coding and LZW coding with 50 random inputs of text"

Submitted by

Subham Chakraborty

17-15-064

Under the Supervision of

Dr. Naresh Babu M.

Assistant Professor

Department of CSE

NIT Silchar

Department of Computer Science and Engineering

National Institute of Technology Silchar

(an Institute of National Importance)

Cachar District, Assam 788010

1. Abstract : This project is based on a comparative analysis of two special algorithms used for compression of messages, namely Huffman coding and Lempel-Ziv-Welch (LZW) coding schemes. The compression algorithms are particularly used for reducing the size of data. It helps in transmission, processing and storing of large files. This compression technique can be either lossy or lossless. In lossy compression there is a loss of data which are less important or unnecessary while in lossless compression there is no loss of information and hence they are reversible, that is, the original message can be retrieved from the coded message. Lossless compression is done by identifying and removing redundancy in the data. This kind of compression is required when there is a need of retrieval of the original message. PNG and GIF use lossless compression schemes. Huffman coding and LZW coding both are lossless compression. This project provides a detailed study of the performance of these two algorithms by taking 50 random inputs.

2. Methodology:

Let us take an example string "AAAABBCBABABCBAADDBDA" for explaining the procedure for both the coding schemes.

2.1 Procedure for Huffman coding:

2.1.1. Count the frequency of each character in the string.

2.1.2. Build a min heap with the number of nodes equal to the number of distinct characters in the string . Each node contains a character, its frequency ,a left and a right pointer.

2.1.3. Extract the two nodes with minimum frequencies and insert a new node with frequency equals to the sum of the extracted nodes.

2.1.4. Make this new node the parent of the two extracted nodes.

2.1.5. Repeat Step3 till the number of nodes is greater or equals to 2.

2.1.6. Extract the top node from the min heap and do preorder traversal.

2.1.7. If a leaf is encountered, store the character at the leaf along with the contents of the array in a map.

2.1.8. After the traversal is complete, traverse the input string once again and output the respective codes stored in the map.

For the given string:

Character	Frequency
A	9
B	7
C	2
D	3

Table 1:Character v/s Frequency

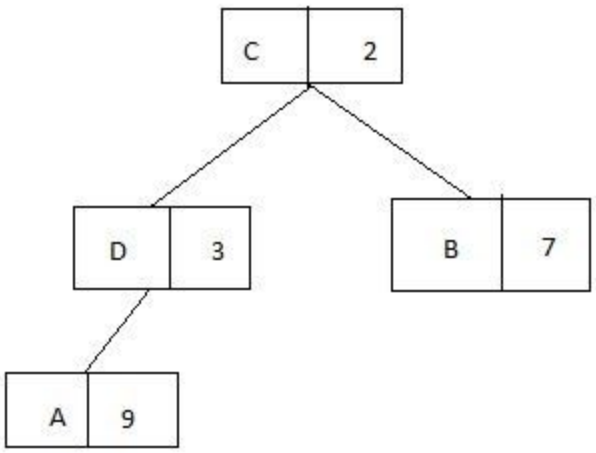


Fig1: Min Heap

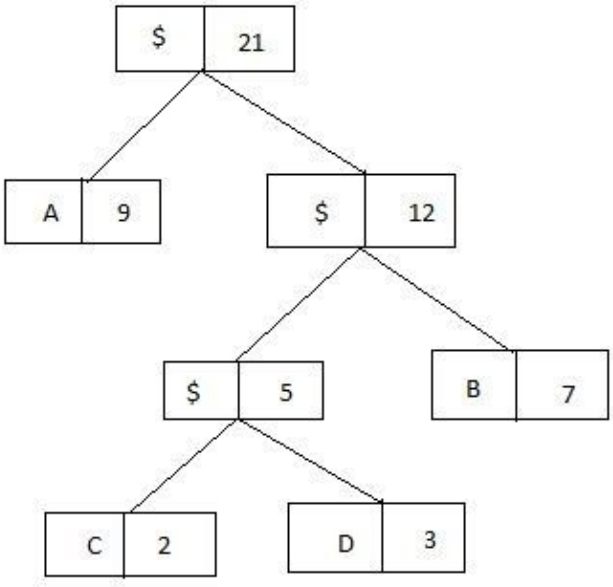


Fig2: Huffman Tree

Therefore the respective codes are:

A-> 0

B-> 11

C-> 100

D-> 101

The coded string is:

0000111110011011011100110010110111101 0

2.2. Procedure for LZW coding:

2.2.1. Create a table with columns- current, next, present/output, dictionary.

2.2.2. Initialize current with the first character in the string.

2.2.3. next=next input character

2.2.4. if current+next is in dictionary

current=current+next

2.2.5. else

output current

add current+next to dictionary

2.2.6. current =next

2.2.7. Repeat steps 2.3, 2.4, 2.5 and 2.6 until the input stream ends

2.2.8. Output current

Current	Next	Present/Output	addtodictionary
A	A	65	AA->256
A	A	Present	-
AA	A	256	AAA->257
A	B	65	AB->258
B	B	66	BB->259
B	C	66	BC->260
C	B	67	CB->261
B	A	66	BA->262
A	B	Present	-
AB	A	258	ABA->263
A	B	Present	-
AB	C	258	ABC->264
C	B	Present	-
CB	A	261	CBA->265
A	A	Present	-
AA	D	256	AAD->266
D	D	68	DD->267
D	B	68	DB->268
B	D	66	BD->269
D	A	68	DA->270
A	-	65	-

Table 2: Explanation of LZW coding

The coded string will be:
65 256 65 66 66 67 66 258 258 261 256 68 68 66 68 65
The coded string contains 16 characters while the original string has 21 characters. Hence there is compression.

3. System Requirements:

3.1 RAM - 8GB
3.2 Processor- Intel Core i5, 8th Generation
3.3 Compiler- Online GDB C++ Compiler
3.4 Language used- C++14

4. Implementation:

Source code:

```
#include<bits/stdc++.h>
using namespace std;
unordered_map<char,string> st;
struct heapNode {
	// One of the input characters
	char val;
	// Frequency of the character
	int freq;
	// Left and right child
	heapNode *left, *right;
	heapNode(char val, int freq){
		left = right = NULL;
		this->val = val;
		this->freq = freq;
	}
};
//structure for comparison between the heap nodes in huffman
struct comp {
	bool operator()(heapNode* l, heapNode* r)
	{
		return (l->freq > r->freq);
	}
};
```

```
void traverse(struct heapNode* root, string s)    //to traverse for the huffman codes
{       //do inorder traversal
        if (!root)
                return;
        if (root->val != '$')
                st[root->val]=s;                    //store the codes for each character in map
        traverse(root->left, s + "0");
        traverse(root->right, s + "1");
}
void huffman(string s){
        unordered_map<char,int>mps;
        //count the frequency of each character
        for(int i=0;i<s.size();i++){
                mps[s[i]]++;
        }
        // create a minheap using cpp stl priority queue
        priority_queue<heapNode*, vector<heapNode*>, comp> minheap;
        //iterate through the map
        for(auto itr=mps.begin();itr!=mps.end();itr++){
                minheap.push(new heapNode(itr->first,itr->second));
        }
        heapNode *left,*right;
        //extract two minimum value nodes and create a newnode
        //make this node the parent of the extracted nodes.
        while(minheap.size()>=2){
                left = minheap.top();
                minheap.pop();
                right = minheap.top();
                minheap.pop();
                heapNode* newnode = new heapNode('$', left->freq + right->freq);
                newnode->left = left;
                newnode->right = right;
                minheap.push(newnode);
        }

        traverse(minheap.top(),"");     //traverse the node for the codes of the character
        cout<<"Huffman coded string: ";
        for(int i=0;i<s.size();i++){
```

```
                cout<<st[s[i]];          //print the huffman codes
        }
        cout<<endl;
}

void lzw_encoding(string s)
{
        unordered_map<string, int> dic;       //dictionary where the codes will be stored
        string curr = "", next = "";
        curr += s[0];                  //curr is initialized with the first character of the string
        int cnt = 256;
        cout<< "LZW code: ";
        for (int i = 0; i < s.size(); i++) {
                if (i != s.size() - 1)
                        next += s[i + 1];
        //if the string containing curr and next is in the dictionary update curr
                if (dic.find(curr + next) != dic.end()) {
                        curr = curr + next;
                }
                else {
               //if curr contains only 1 character its respective ASCII value can be returned
                        if(curr.size()==1){
                                int x=curr[0];
                                cout<<x<< " ";
                        }
                        //otherwise return the value stored in the dictionary
                        else
                                cout<<dic[curr]<< " ";
                        dic[curr + next] = cnt;
                        cnt++;
                        curr = next;
                }
                next = "";
        }
        cout<<endl;
}
int main()
{
  clock_t starttime, endtime1,endtime2;
```

```
//start timer
 starttime=clock();
 huffman("AAAABBCBABABCBAADDBDA");
// find the time after huffman coding
 endtime1=clock();
 lzw_encoding("AAAABBCBABABCBAADDBDA");
 // find the time after lzw coding
 endtime2=clock();
 double t1=double(endtime1-starttime)/double(CLOCKS_PER_SEC);
 double t2=double(endtime2-endtime1)/double(CLOCKS_PER_SEC);
 cout<<"time taken by huffman: "<<t1<<"\n"<<"time taken by lzw: "<<t2 ;
 return 0;
}
```

5 Results:

The time comparison required for different inputs in both the coding schemes are as shown below:

Serial No.	String Length	String Input	Time reqd. in Huffman Coding	Time reqd. in LZW Coding
1	5	BCCCC	0.000071	0.000002
2	6	GFDSER	0.000164	0.000031
3	7	MXFGGXX	0.000099	0.000078
4	8	SUBHAMBA	0.000140	0.000048
5	9	PJJHGGDGD	0.000257	0.000098
6	10	MPERTYUVUI	0.000198	0.000087
7	11	SDHGGTHJKIO	0.000232	0.000049
8	12	BBBNNNUIOGAS	0.000155	0.000037
9	13	PPPPPUIOSSSSS	0.00012	0.000053

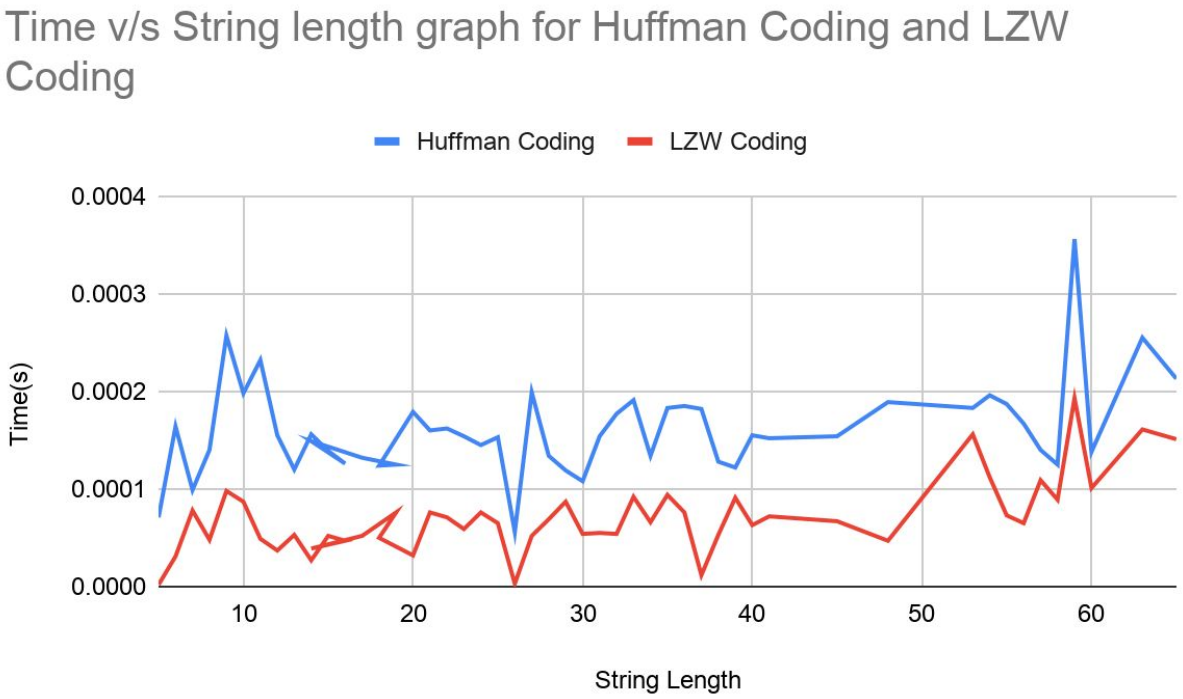
10	14	LLLLLLJMARKBVC	0.000156	0.000027
11	15	MONFDFGHERFDESW	0.000139	0.000052
12	16	HAHAHAJKLMNBVCAA	0.000126	0.000047
13	14	MMMMMMFRTYMMMM	0.000149	0.000039
14	17	MONFDFGKKKKKKK***	0.000132	0.000052
15	19	NNNNNNNRTYUMNNANNN	0.000125	0.000076
16	18	MSKJNKJSNCKJNMKKKK	0.000124	0.000031
17	20	NSHBSXHBJJJJKKKKJJJJ	0.000179	0.000032
18	21	JSNSNKCSCKJMKLMMMSCS	0.000160	0.000076
19	22	JJKSJXSJMKKKKKKKKKXXXX	0.000162	0.000071
20	23	JJJJJJHSHGGSGGSGCYSGHHH	0.000154	0.000059
21	24	IIIIIYYHHSGGSGGSGGSGGHHYY	0.000145	0.000076
22	25	AAAABBCBABABCBAADDBDACBBC	0.000153	0.000065
23	26	OOOOOOOOOOOOOOOOOOOOOOOOOO	0.000056	0.000003
24	27	NNNNNNNNNNNNNNNNNNNNNNNNNN	0.000199	0.000052
25	28	SSSSSSJJJJNSKJNJSNJNKKNNN	0.000134	0.000069
26	29	MMKMKMKMKMKMKMKMKSKSKKMMKKKSK	0.000119	0.000087

27	30	JJSBHSBHHHHHHSHHSHHSHHSHHSHSSH	0.000108	0.000054
28	31	HHGSFFSFFSFFTTXVGVVHBHBFFTSTXV	0.000154	0.000055
29	32	JJJSUUSYYSYYBHBBHBBJJJJUJBSBSSYY	0.000177	0.000054
30	33	JJSHSHHSHHSHHSHHSHHBBBHBHBRRSTHH	0.000191	0.000092
31	34	TTTTTTTTTTTTTSVVVVVVVVVVRRRRRTTTT	0.000134	0.000066
32	35	YYYXVBHTWQLBBWTTEGGEGGEGMMKLFGEGGEG	0.000183	0.000094
33	36	HHDGGDHHDHHDHHDHHDHHDHHDGJJKWWWGDGG	0.000185	0.000076
34	37	HHDHHDGGDGGDGGDGGDGGDGG***D**C*CCZZXZ	0.000182	0.0000118
35	38	HHSMMMMSMMSMMSMMSMMMNNNNMSMSMMNSHHHS	0.000128	0.000053
36	39	UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU	0.000122	0.000091
37	40	KKKKKDKKDKKDKKDKKDKKDKKKLLLKLKGGDKLDKLL	0.000155	0.000063
38	41	KKKKKFFFKKKDKKDKKDKKDKKDKKDKKKLKLGHOPPOFP	0.000152	0.000072

39	45	NNNNNNDNNDNNDNNDNNDNNDNNDNN…...FF..THEKGHTHE	0.000154	0.000067
40	48	BAMUNBAMUNBAMUNBAMUNBAMUNBAMNUBMNAUBMNUABB….BA	0.000189	0.000047
41	53	entries initialized to single characters for everyone	0.000183	0.000156
42	54	dhsksbdkdbdvdnjduddd dg s dbdvdhbd dnckc cjjllldnoii	0.000196	0.000112
43	55	JJJJDJJDJJDJJDJJDJJDJJDJJDJDJJDJJDJJDJJDFDGDDGDGGJJGJ	0.000187	0.000073
44	56	JJDJJDJJDJJDJJDJJDJJDJJDJJDJJJJJJJJJJJJJJDDDDDJJJDIIIIJD	0.000167	0.000065
45	57	LLLAAAAAAAAAAAAAAAAAAAAAAAAAALOLLLLLLLL…….LLOAOALALAO	0.000140	0.000109
46	58	HHHHHHHHHHHHHHHHHHS……...SSSHHHHHHHSHSHSHHS	0.000125	0.000089
47	59	ABCDEFGHIJKL...MNOPQRSTUVWXYZABCDEFGH_IJKLMNOP__QRSTUVWXYZ.	0.000356	0.000193
48	60	JJJJJJSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSRRRRRRBBBBRBRBRBBBRRBB	0.000138	0.000101
49	63	JJSJJSJJSJJSJJSJJSJJSJJSJJ...Abcdefffcdefgghgjjjewreeeo	0.000255	0.000161

		opqrsttu		
50	65	dea....neve...fea..if..i...m.here ……...ev..r..thngisnear.okbye	0.000213	0.000151

Table 3: Time required for generating codes in Huffman and LZW coding



Observation from the above graph are as follows:

- The time required for both the coding mainly depends on the pattern in the and not on the string length. It is because in Huffman coding the code length depends on the number of unique characters in the string. More is the number of unique characters, more is the time required. On the other hand, in LZW coding the time required is dependent on the repetition of the patterns. More is the repetition, less will be the time required.
- For any string length Huffman Coding takes more time compared to LZW coding. It is because in LZW coding the string is coded in one pass, while in Huffman coding at first the string is traversed to count the frequency of each character. Then the heap is created and traversed for obtaining the respective codes.

Worst case time complexity:

Huffman coding: O(nlogn)

LZW coding: O(n)

6. Conclusion:

The key differences between the two coding schemes are:

- LZW coding does not require any prior information about the input stream whereas Huffman coding requires prior information about the input stream. Hence we cannot use Huffman coding when there is a continuous input.
- In the LZW scheme the string can be compressed in one pass whereas it is not possible in Huffman coding scheme.
- When the frequency of each character is very high in the input string, Huffman coding gives far better compression than LZW coding. Otherwise LZW provides better results.

From the above discussion, it is clear that both the coding schemes can be used depending on the input string. But LZW coding promises better performance for any random input stream.

7. References:

i)http://web.mit.edu/6.02/www/s2012/handouts/3.pdf

ii)https://www2.cs.duke.edu/csed/curious/compression/lzw.html

iii)http://michael.dipperstein.com/lzw/