Mini Project on

Comparative study of Huffman coding and LZW coding with 50 random inputs of text

Submitted by
Padmanabha Bhattacharjee
17-1-5-014

Under the Supervision of
Dr Naresh Babu M.
Assistant Professor
Dept of CSE
NIT Silchar
Department

Department of Computer Science and Engineering
National Institute of Technology Silchar
(an Institute of National Importance)
District:- Cachar, Assam, PIN:- 788010

Table of Contents

1) Abstract

The aim of this project is to analyse the compression techniques LZW and Huffman and prepare a comparative report based on observations. Both the techniques are lossless compression techniques. In lossless compression techniques data is not lost while compression. The other type of compression technique i.e. lossy compression technique involves loss of redundant data. Lossy compression techniques include methods such as DCT, Vector Quantisation etc. Through this project an attempt has been made to better understand LZW and Huffman lossless techniques and how they compare.

Data which is uncompressed occupies a large amount of space and is not very efficient. Downloading files which are very large in size is very impractical. Devices with limited storage cannot store files which are large. This issue is resolved with the help of compression techniques. Compression techniques generally make use of duplicate data to compress files. Duplicate data unnecessarily take up space. This space is what is freed when compression techniques are applied. Every compression technique has its own method of compressing data. At the same time, there must be a way to decompress data in order to be able to read it. For this purpose, decompression techniques are also present. Decompression techniques mainly expand the condensed form of the data . Every compression technique has its own unique decompression technique which are also known

as encoding and decoding.

These techniques can be used on all kinds of files be it image file or video file or any sort of text file. The comparisons in this project will be made using text files.

2) Methodology

2.1) LZW coding

LZW (Lempel–Ziv–Welch) coding technique is a dictionary-based lossless compression technique which is largely used in compressing GIFs and also PDFs. The Unix uses this technique forcompression purposes. It is a simple algorithm which uses recurring patterns for its compression method. The larger the number of repeated words the greater the efficiency of the algorithm.

2.1.1) Algorithm

In LZW coding, a codetable is at first constructed. The codetable consists of all the characters mapped to some value. The entire text to be compressed is then read.
At each stage if any word that is in the codetable is encountered it is replaced with the code or value of the word in accordance with the codetable. If the word is not in the codetable the word is included in the codetable and assigned a value or code. Whenever the same word is encountered it is replaced by that code. LZW basically identifies repeated sequences and maps them to some value in the codetable.

The LZW codetable generally uses 4096 as the number of entries. 256 characters are Initially mapped based on ASCII values. As new strings are encountered they are mapped to subsequent values. The new strings are essentially a combination of existing strings.

Example:-

Let us take the string ‘ABABAB’

For simplicity, let A-> 1 and B->2

i) ‘A’ is already mapped.

ii) ‘AB’ is not mapped . Hence , we output 1 the code value of ‘A’. ‘AB’ is assigned the code value 3.

iii) ‘B’ is already mapped.

iv) ‘BA’ is not mapped. Hence, we output 2 and ‘BA’ is assigned value 4.

v) ‘A’ is already mapped.

vi) ‘AB’ is also mapped.

vii) ‘ABA’ is not mapped. So, we output 3 and assign ‘ABA’ value 5.

viii) ‘A’ is already mapped.

ix) ‘AB’ is already mapped.

x) End of text is read and 3 is printed again.

xi) The encoded text is 1233.

2.1.2) Pseudo code

Step 1: Construct a codetable where each character is mapped to a value(ASCII value)

Step 2: Initialize an empty string S

Step 3: Repeat Steps 4 and 5 till end of input text

Step 4: Read a character and add it to the string S

Step 5: If the string is present in the codetable, goto Step 4

Else add the string to the codetable and map it to a value. Remove the last added

Character in S and print the value of the string S mapped to it

in the codetable. Set S as the current character and goto Step 4

Step 6: Print the value of the String S

2.2) Huffman Coding

Huffman coding is another lossless compression technique. It makes use of prefix codes wherein the code assigned to one character is not the prefix of any other character. For Huffman coding bit streams are used to represent characters. At first a Huffman tree is built and then traversing along the tree bit sequences are assigned to characters. The characters are then replaced by the bit sequences.

2.2.1) Algorithm

The frequencies of the characters are first calculated and nodes are created based on those frequencies. The Huffman tree is then built by choosing the two nodes with the smallest frequencies and adding those two nodes to another node whose frequency is the sum of frequencies of the two nodes and the two nodes become children to the newly created node. The two nodes are replaced by the newly created node. In short, a min-heap is created from the nodes.

The Huffman tree is traversed and the left edge of every node is given value 0 and right node is given value 1. The leaf node consists of the single characters and the value of each character is the bit sequence in the path from the root to that node. The characters of the text are replaced by the bit sequences of the corresponding characters.

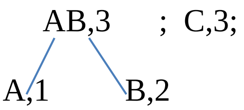
Example:-

Let us take the string ‘ABBCCC’

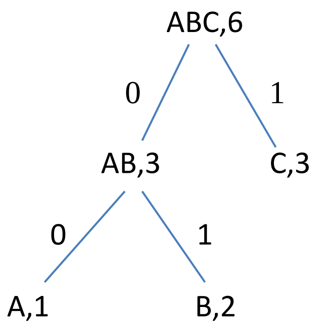
i) The leaf nodes are:-

A,1 ; B,2 ; C,3 ;

ii) Creating internal node using priority queue, we get



iii) Creating the full min-heap :-



iv) Therefore, A-> 00, B-> 01, C-> 1

v) The resulting text is 000101111

2.2.2) Pseudo Code

Step 1: Initialise the leaf nodes with the individual characters and their frequencies as their respective values. Insert the nodes in a priority queue with higher priority given to nodes with lesser frequencies.

Step 2: Repeat Step 3 till there is only one node.

Step 3: Pick two nodes with lowest frequencies and create a new node with frequency equal equal to the sum of the picked nodes’ frequencies. Set the newly created node as the parent of the picked nodes. The two nodes are removed from the priority queue and the *newly created node is added.*

Step 4: Mark the node that is left as root node R.

Step 5: Traverse the tree with R as root node and mark the edge that connects the left child for every node as 0 and the right child edge as 1.

Step 6: Calculate the value of each leaf node(character) as the bit sequence along the path from the root node to the leaf.

Step 7: Replace the input text with the bit sequence of each character.

3) System Specifications

3.1) Hardware

i) Operating System : Microsoft Windows 10 Pro

ii) System Model : HP Laptop 15-da0xxx

iii) Processor : Intel(R) Core(TM) i3-7100U CPU @ 2.40GHz

iv) Installed Memory : 8.00 GB

v) System type : 64-bit Operating System, x64-based processor

vi) Inbuilt Graphics : Intel(R) HD Graphics 620

Memory : 4181 MB

vii) Graphics Card : GeForce GTX 1050 (2GB)

3.2) Software/Tools

i) Codeblocks

ii) Microsoft Visual Studio

iii) Microsoft Word

iv) Microsoft Excel

v) MinGW GNU for Windows(64-bit)

4) Implementation

4.1) C++ Implementation of LZW Coding

Here is the C++ code :-

/***** LZW Coding *****/

```
#include<bits/stdc++.h>
//#include <ext/pb_ds/assoc_container.hpp>
//#include <ext/pb_ds/tree_policy.hpp>
#define ll long long
#define pb push_back
#define mod 1000000007
#define ff first
#define ss second
#define pi 3.1415926535
#define endl '\n'
using namespace std;
struct custom {
    static uint64_t splitmix64(uint64_t x)
    {
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
```

```
    }
    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM =
chrono::steady_clock::now().time_since_epoch().count();
        return splitmix64(x + FIXED_RANDOM);
    }
};
//using namespace __gnu_pbds;
//typedef tree<ll,null_type,less_equal<int>,rb_tree_tag,tree_order_statistics_node_update>
indexed_set;
const ll inf=1e18;
/*ll power(ll x,ll y)
{
    ll res=1;
    while(y>0)
    {
        if(y%2==1)
            res=((res)*(x))%mod;
        x=((x)*(x))%mod;
        y=(y>>1);
    }
    return res;
}*/
// code starts here
int main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(0);
```

```
cout.tie(0);
freopen("input.txt","r",stdin); // opening input text file for reading
freopen("output.txt","w",stdout); // opening output file for printing encoded text as
                                            // output in output file

string s,s1;
cin>>s;
ll i,a,b,c,d,x=0,y=256;
char ch;
unordered_map<string,ll>mp,mrk;
for(i=0;i<=255;i++) // mapping all 255 characters
{
   string s2;
   ch=i;
   s2.pb(ch);
   mrk[s2]=1;
   mp[s2]=i; // mapping characters in the codetable to ASCII values
}
for(i=0;i<s.size();i++)
{
   s1.pb(s[i]);
   if(!mrk[s1]) // checking if string is present in codetable
   {
      mrk[s1]=1;
      mp[s1]=y; // mapping value to the new string
      y++;
      s1.pop_back();
```

```
            cout<<mp[s1]; // printing value of the string
            string s2;
            s2.pb(s[i]);
            s1=s2; // setting s1 equal to the current character
            continue;
        }
    }
    cout<<mp[s1]; // printing the last value
}
```

4.2) C++ Implementation of Huffman Coding

Here is the C++ code of Huffman Coding,

```
                    /***** Huffman Coding *****/

 #include<bits/stdc++.h>
//#include <ext/pb_ds/assoc_container.hpp>
//#include <ext/pb_ds/tree_policy.hpp>
#define ll long long
#define pb push_back
#define mod 1000000007
#define ff first
#define ss second
#define pi 3.1415926535
#define endl '\n'
using namespace std;
struct custom {
```

```
    static uint64_t splitmix64(uint64_t x)
    {
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }
    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM =
chrono::steady_clock::now().time_since_epoch().count();
        return splitmix64(x + FIXED_RANDOM);
    }
};
//using namespace __gnu_pbds;
//typedef tree<ll,null_type,less_equal<int>,rb_tree_tag,tree_order_statistics_node_update>
indexed_set;
const ll inf=1e18;
/*ll power(ll x,ll y)
{
    ll res=1;
    while(y>0)
    {
        if(y%2==1)
            res=((res)*(x))%mod;
        x=((x)*(x))%mod;
        y=(y>>1);
    }
    return res;
```

```
}*/
// Traversing the Huffman tree and allocating bit sequence to leaf nodes i.e. characters
void dfs(vector<ll>v[],ll node,string s1,unordered_map<ll,string>&mp2)
{
    mp2[node]=s1;
    ll x=0;
    char ch='0';
    for(auto u : v[node])
    {
        ch=ch+x;
        x=!x;
        s1.pb(ch);
        dfs(v,u,s1,mp2);
        s1.pop_back();
    }
}
int main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    freopen("input.txt","r",stdin); // opening input text file for reading
    freopen("output1.txt","w",stdout);  //opening output file for printing encoded text as output
                                               //  in output file
    string s;
    cin>>s;
    priority_queue<pair<ll,ll>,vector<pair<ll,ll> >,greater<pair<ll,ll> > >pq; // priority
```

```
                                         // queue for building Huffman tree

ll n,i,a,b,c,d,x,y,r=0,k;
n=s.size();
vector<ll>v[4*n+1]; // for creating adjacency list
unordered_map<char,ll>mp;
unordered_map<ll,ll>mp1; // for storing frequencies of characters
unordered_map<ll,string>mp2;
for(i=0;i<n;i++)
{
   if(!mp[s[i]])
   {
      r++;
      mp[s[i]]=r;
   }
   mp1[mp[s[i]]]++;
}
for(i=1;i<=r;i++) //  creating leaf nodes and storing in priority queue
{
   pq.push({mp1[i],i});
}
while(pq.size()>1) // creating Huffman tree
{
   x=pq.top().ss,a=pq.top().ff;
   pq.pop();
   y=pq.top().ss,b=pq.top().ff;
   pq.pop();
```

```
        r++;
        v[r].pb(x),v[r].pb(y);
        pq.push({a+b,r});
    }
    string s1="";
    ll node=pq.top().ss;
    dfs(v,node,s1,mp2);
    for(i=0;i<s.size();i++)
        cout<<mp2[mp[s[i]]]; // Printing the encoded text
}
```

5) Experimental Results

The algorithms of LZW coding and Huffman coding were run on 50 different inputs of text. The inputs of text were generated purely randomly and the compression texhniques were run on them. The results are compared in this section both in tabular and graphical form. The observations made over 50 inputs of test have been stored. The compression ratio of each test has been calculated and an average compression ratio has also been calculated.

The observations have also been represented in a graphical form for easy comparison between the techniques. At last, experimental results have been derived from these observations and have been presented here.

5.1) Tabular Observations

The algorithms have been run on similar input tests and result has been documented in this section in tabular form.

Observation table for LZW coding :-

Input Text Size(in bits)	Output Text Size(in bits)	Compression Ratio
1600	528	3.0303
1680	372	4.51613
1760	2520	0.698413
1840	684	2.69006
1920	1932	0.993789
2000	492	4.06504
2080	1488	1.39785
2160	516	4.18605
2240	552	4.05797
2320	1452	1.5978
2400	804	2.98507
2480	996	2.48996
2560	804	3.18408
2640	804	3.28358
2720	3024	0.899471
2800	720	3.88889
2880	2064	1.39535
2960	3300	0.89697
3040	732	4.15301
3120	1164	2.68041
3200	852	3.75587
3280	2052	1.59844
3360	1296	2.59259
3440	2304	1.49306
3520	1176	2.9932
3600	792	4.54545
3680	924	3.98268
3760	4704	0.79932
3840	1248	3.07692
3920	2316	1.69257
4000	840	4.7619
4080	1284	3.17757
4160	1200	3.46667
4240	4716	0.899067
4320	1500	2.88
4400	6288	0.699746
4480	996	4.49799
4560	1176	3.87755
4640	1080	4.2963

4720	1080	4.37037
4800	1416	3.38983
4880	1356	3.59882
4960	1272	3.89937
5040	1332	3.78378
5120	1140	4.49123
5200	1248	4.16667
5280	1512	3.49206
5360	1308	4.09786
5440	1368	3.97661
5520	1848	2.98701

Observation table for Huffman Coding :-

Input Text Size(in bits)	Output Text Size(in bits)	Compression Ratio
1600	929	1.72228
1680	979	1.71604
1760	1020	1.72549
1840	1068	1.72285
1920	1116	1.72043
2000	1173	1.70503
2080	1213	1.71476
2160	1255	1.72112
2240	1313	1.70602
2320	1351	1.71725
2400	1402	1.71184
2480	1447	1.71389
2560	1500	1.70667
2640	1536	1.71875
2720	1594	1.7064
2800	1629	1.71885
2880	1693	1.70112
2960	1737	1.70409
3040	1787	1.70118
3120	1834	1.7012
3200	1883	1.69942
3280	1925	1.7039
3360	1962	1.71254
3440	2012	1.70974
3520	2052	1.7154
3600	2108	1.70778
3680	2156	1.70686
3760	2210	1.70136

3840	2256	1.70213
3920	2306	1.69991
4000	2355	1.69851
4080	2395	1.70355
4160	2436	1.70772
4240	2480	1.70968
4320	2532	1.70616
4400	2584	1.70279
4480	2627	1.70537
4560	2677	1.7034
4640	2725	1.70275
4720	2783	1.69601
4800	2820	1.70213
4880	2871	1.69976
4960	2916	1.70096
5040	2966	1.69926
5120	3011	1.70043
5200	3058	1.70046
5280	3107	1.69939
5360	3158	1.69728
5440	3211	1.69418
5520	3251	1.69794

5.2) Graphical Observations

A graph is plotted between the Input Text Size and Output Text Size over 50 tests of input.

The data of the points will be in accordance with the above data presented in tabular form.

This graph will provide us a method to compare the test results in a lucid manner.

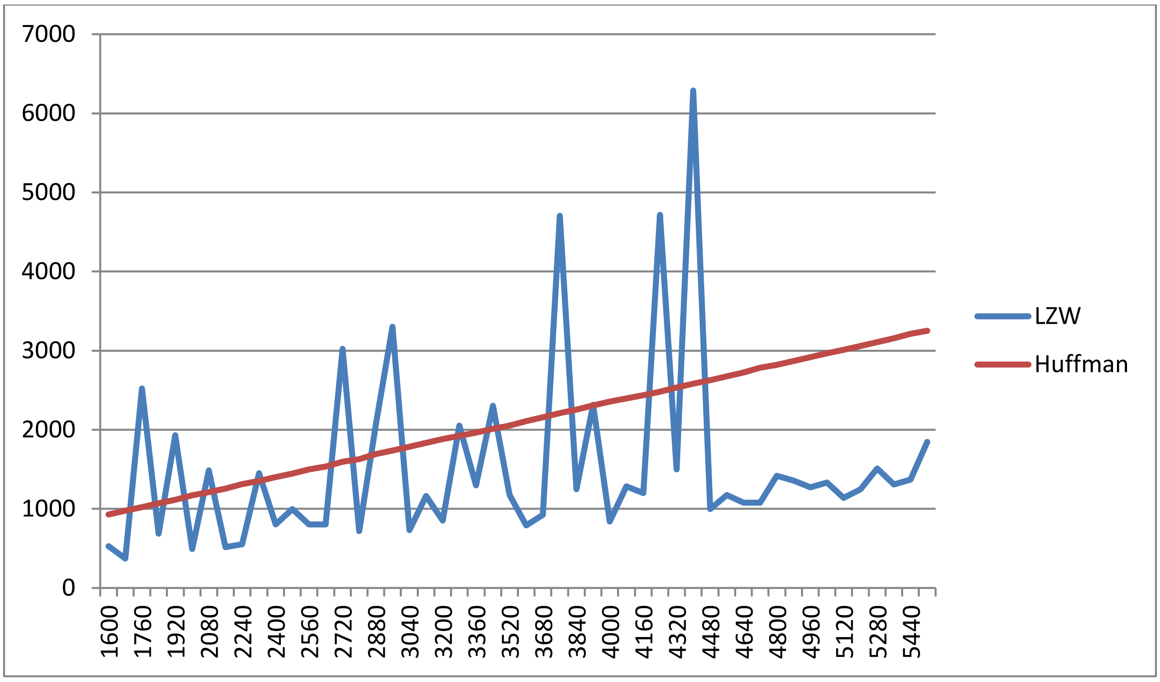
The graph will be a line graph. The line graphs of both the compression techniques will

Be plotted on the same graph space in order to ease comparisons.

Along X-axis :- Input Text characteristics is plotted

Along Y-axis :- Output Text characteristics is plotted

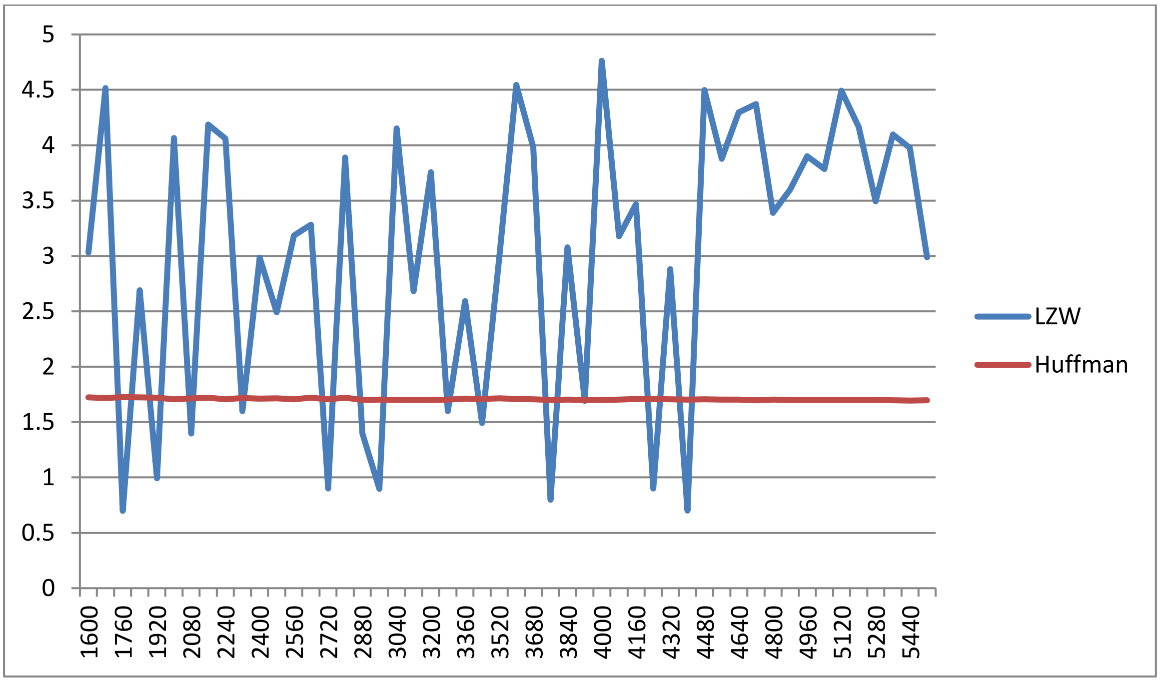
The plotted graph between Input Text Size and Output Text Size is given below :-



X-Axis:- Input Text Size (in bits)

Y-Axis:- Output Text Size (in bits)

Input Text Size versus Compression Ratio graph is plotted below :-



X-Axis:- Input Text Size (in bits)

Y-Axis:- Compression Ratio

5.3) Results

Compression Ratio is defined as the ratio between Input Text Size and the Output Text Size. The average compression ratio for both LZW and Huffman coding techniques have been calculated with the help of tabulated data.

Average Compression Ratio for LZW coding is :- 3.00861

Average Compression Ratio for Huffman coding is :- 1.70684

From this data, it can be observed that on an average LZW works better than Huffman. Inspecting the Compression Ratio graph, we see that the ratio for LZW fluctuates a lot in the initial stages and as the input text size increases the fluctuations start decreasing. The compression ratio for Huffman coding, however, remains constant through the varying Input text sizes.

6) Conclusion

From the observations, it can be concluded that the LZW technique depends heavily on the Patterns existing in the text file. The more the patterns, the better the algorithm will work. The Huffman coding scheme does not, however, depend on the pattern and hence gives a consistent compression ratio. The graph of LZW can be explained on the basis of occurrence of patterns. As the size of text grows, recurring patterns increase and the consistency of compression ratio rises. Hence, the curve is somewhat consistent towards the end.

Therefore, LZW coding technique should be used when repetitions of text is expected in the text file as it relies on recurring patterns. For purely random texts, use of Huffman coding is better as the behaviour of Huffman coding is predictable and does not make use of recurring patterns. In general, LZW is better as it gives a much higher compression ratio. So, the use of compression techniques also depends on the type of file to be compressed.

7) References

i) GeeksforGeeks

ii) Data Structures Using C by Reema Thareja

iii) Competitive Programmer’s Handbook by Antti Laaksonen

iv) Algorithms Unlocked by Thomas H. Cormen