LOSSLESS DATA COMPRESSION ANALYSIS OF COMPUTER ALGORITHMS

Nitheesh Edla
Department of computer science
University of North Texas
(UNT) Denton, Texas, USA
nitheeshedla@my.unt.edu

Pragnya Gannapureddy
Department of computer science
University of North Texas
(UNT) Denton, Texas, USA
pragnyagannapureddy@my.unt.edu

1. Objective

Loss less data compression is a technique that reduces the size such that a decompression function can restore the original file exactly with no loss of data. The main objective of the project is to decrease the utilization of resources, such as transmission bandwidth or disk space. The steps included in this are encoding the input file and convert it into binary codes and then again decoding it. Using greedy approach, we are implementing this loss less data compression algorithm.

2. Motivation:

We discovered that lossless data compression is extremely useful when compressing data by encoding and decoding, and we also learned about a few applications that use this technique. We are attempting to make the file as small as possible in comparison to the applications, and this project will require a significant amount of learning.

3. Background :

After looking at a variety of compression techniques such as Shannon-Fano Coding, Arithmetic Coding, Huffman Coding, and a few audio compression algorithms, we decided to create an algorithm that is similar to Huffman Coding but uses fewer bytes after compression. We aimed to achieve our desired outcome by doing the polar opposite of Huffman coding in this approach.

4. Application of Algorithms :

One quite simple approach of compression is run-period encoding, in which massive runs of consecutive same data values are changed with an easy code with the data value and period of the run. This is an instance of lossless records compression. It is regularly used to higher use disk area on workplace computers, or higher use the relationship bandwidth in a pc network. For symbolic data like spreadsheets, text, executable programs, etc., loss lessness is important due to the fact converting even a single bit cannot be tolerated (besides in a few restricted cases).

Generally, for lossless data compression, we use Huffman coding which is a greedy approach technique where each character gets ASCII value and that value is converted into 8-bit binary codes. But the disadvantage of using Huffman coding is it is slower for reading and writing files. Our application has an improvised version of Huffman, making faster reading and writing input files.

5. ALGORITHM :

We'll use the Greedy approach to create an algorithm that performs four jobs, each of which will help us reach our aim of lossless data compression and encoding.

The four tasks are as follows:
1. Obtain and save the frequencies of all the distinct characters found in the file.
2. Create a tree and store all of the characters according to the frequency created before, assigning binary codes to each leaf and parent node.
3. Combine all of the binary values along the way to encode all of the characters in the tree. (For example, if we have a character at the third level and on the right, the binary value of that character would be 01 (if the character's parent is a left node) or 11 (if the character's parent is a right node) (if the parent is also a right node).
4. Rearrange the binary values by decoding them into characters.

The below function does the frequency calculation and build a tree:

```
void buildLosslessCompressionTree(string text){
// Count the number of times each
character(frequency)
//appears and save the data in to a map
 unordered_map<char, int> fre;
int i=0;
        while(text[i]!='\0')
        {
                        fre[text[i]]+=1;
                        i++;
        }

// Creating a priority queue 'priority' to store the
present //nodes from the tree;
priority_queue<Node*, vector<Node*>, comp>
priority;

// Creating a leaf node for each character and
pushing it in to the priority queue created earlier.
for (auto item: fre) {
priority.push(getNode(item.first, item.second,
nullptr, nullptr));
        }
        while (priority.size() != 1)
        {
// From the queue, remove the two nodes which
has the highest priority (lowest frequency).
Node *left_node = priority.top(); priority.pop();
Node *right_node = priority.top();
        priority.pop();

//creating a new node, that is created by adding
up the //frequencies of the earlier popped nodes
and add the new //node to the existing priority
queue.
int sum = left_node->fre + right_node->fre;
priority.push(getNode('\0', sum, left_node,
right_node));
        }
// root stores a pointer to root of Huffman Tree i.e
top node of the priority queue
        Node* root = priority.top();
// traverse the Tree and store Codes in a map .
Also prints them
unordered_map<char, string> lossless;
encode_string(root, "", lossless);

cout << "\nGiven string is :" << text << '\n';
        cout << "Binary Codes for the characters
are :\n" << '\n';
        for (auto item: lossless) {
                        cout << item.first << " " <<
                item.second << '\n';
        }
        // printing encoded string
        string str = "";
        for(int i=0;text[i]!='\0';i++)
        {
                str=str+lossless[text[i]];
        }
   cout << "";
    ofstream f;
    f.open("example.txt");
  f << str;
```

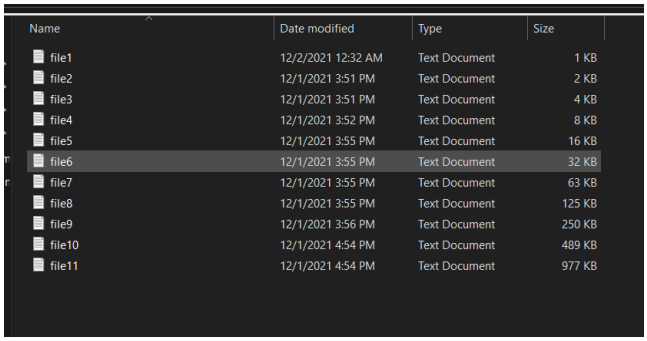
```
 cout<<"written to file";
 f.close();
        cout << "\nEncoded string is :\n" << str
<< '\n';

}
        }
        // printing encoded string
        string str = "";
        for(int i=0;text[i]!='\0';i++)
        {
                str=str+lossless[text[i]];
        }
   cout << "";
    ofstream f;
    f.open("example.txt");
 f << str;
 cout<<"written to file";
 f.close();
        cout << "\nEncoded string is :\n" << str
<< '\n';

}
```

6. Experiments :

6.a Datasets:

We have used several datasets of size 2^0 to 2^11, and the type of the file is text, following is the snapshot of the files we have used in the project.

List of input files:



Name	Date modified	Type	Size
file1	12/2/2021 12:32 AM	Text Document	1 KB
file2	12/1/2021 3:51 PM	Text Document	2 KB
file3	12/1/2021 3:51 PM	Text Document	4 KB
file4	12/1/2021 3:52 PM	Text Document	8 KB
file5	12/1/2021 3:55 PM	Text Document	16 KB
file6	12/1/2021 3:55 PM	Text Document	32 KB
file7	12/1/2021 3:55 PM	Text Document	63 KB
file8	12/1/2021 3:55 PM	Text Document	125 KB
file9	12/1/2021 3:56 PM	Text Document	250 KB
file10	12/1/2021 4:54 PM	Text Document	489 KB
file11	12/1/2021 4:54 PM	Text Document	977 KB

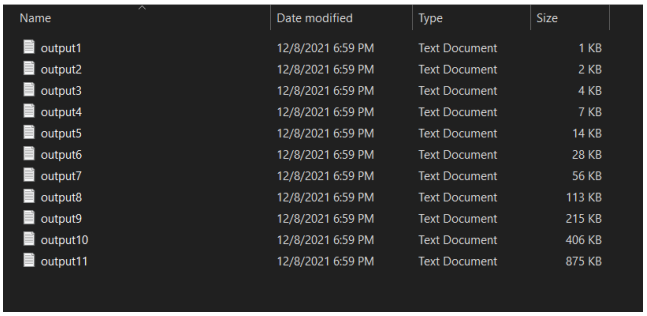
File data:

```
File  Edit  Format  View  Help
HYksVCNnYPzi5QSG3NhtowYQW+Y5JdZKznnw55Ku3w2eKPMc5EIPpA92UVEm4rrb0FJGPyvygN2
3L8fuUpLscTib2D4/n2gPVo0VQbe0a/LlpsZIFQEr/pFKK4xJBTHJ9zph9IzzQ3qkR/aBhw8Dte
mjcuzI0nE7SdPbxOrJhBCFiXV51eXNFnL1Cr4M2vB7O3KwaI0HT8QkL5JtVX4s1W2XBidisidW7
HXENI5DsioB7yx6Jz9MhgruIK8wGujeAuVpWi8gd3qmhrayqhiifC4JRMklxAhkK3052VwR1k6O
UPx+tMyHpvfQAT9Ku8zinQLVDzuTx6su+JBGx5nZ6BWXrsopt8BdR8g8h0rSt5onNjIbPqjS17V
nCecnuFchz/LKN2ldAOdTp9ecrIlyQ/8XfxJhSibkodXpfoFJt5Z/LLdeNlCGbo05k1MUjbSqoQ
6EyaNeSeYj3ePcN6S4IAiL/NlFoZjFNdUOI7Ab3x7kUC1LvbZpvRat2hlvEXOL1HaV9ISw6BLRt
k+cCrVWf2KrBpG/O7mlHDTdHrCUqYG83WSSWZafH+T3Jxn4WxthPUFWKtw0jGwTeCbBY739ZWZH
/RATejWRsRUfCUu93l/Ehp4WM02le7mdiuoRtO/xnEjoeFs1pZJ4AXW+G4ojbSCEmdWXV81tVlw
Z+0WFA0xHesbE7HYy+yefMA6HMGU9dbz/kSLm8mHEBXXD+45dRUkp/Bv0u0p8cCeyaAV7V0MIr/
DN/Do4gwyOIuHbvQz0/8f7CVGewi9msHQaVC8sk4dNroWNq0sgnBrxJfat4dCB8J71h6Eje4ixG
```

6.b Purpose of each experiment:

The purpose of each and every experiment is to get the desired result with less time consuming and we can see that as the size of the file is increasing the time taken for executing is also increasing the size of the output file is also, we can see that the new file generated is encoded into binary and can be seen a slight decrease in size but sometimes based on the number of characters if may differ and for large files the time taken for the execution is so long.

List of output files:

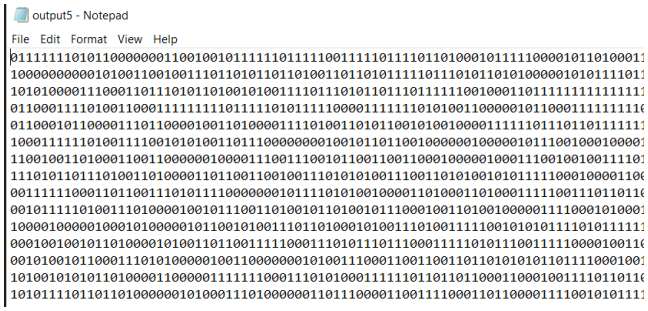


Name	Date modified	Type	Size
output1	12/8/2021 6:59 PM	Text Document	1 KB
output2	12/8/2021 6:59 PM	Text Document	2 KB
output3	12/8/2021 6:59 PM	Text Document	4 KB
output4	12/8/2021 6:59 PM	Text Document	7 KB
output5	12/8/2021 6:59 PM	Text Document	14 KB
output6	12/8/2021 6:59 PM	Text Document	28 KB
output7	12/8/2021 6:59 PM	Text Document	56 KB
output8	12/8/2021 6:59 PM	Text Document	113 KB
output9	12/8/2021 6:59 PM	Text Document	215 KB
output10	12/8/2021 6:59 PM	Text Document	406 KB
output11	12/8/2021 6:59 PM	Text Document	875 KB

Characters are encoded as below

```
Given string is :data compression algorithm.
Binary Codes for the characters are :

l 11111
h 11110
t 1110
p 11010
i 1100
n 10111
m 1000
  0101
e 0100
o 001
d 10110
r 000
g 10011
a 011
```

Output file data:

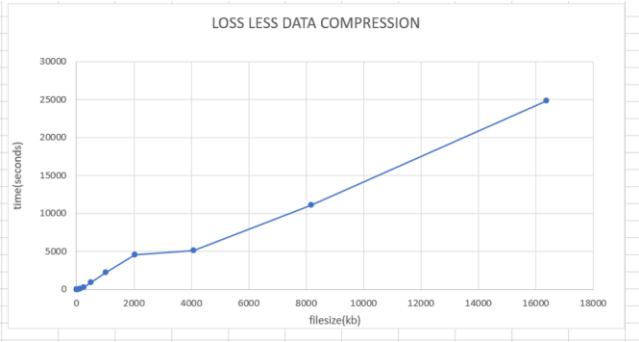


File size(kb)	Time
1	0.006
2	0.0015
4	0.039
8	0.174
16	0.707
32	3.112
64	13.672
128	72.41
256	241.6
500	941.2
1024	2207.9
2048	4521.2
4096	5142
8192	11087.21
16384	24856.03

6.c Experimental results:

The complexity of the project is Big O of (log n)^2

The execution time for the project in different experiments is as follows:



As you can see from the above table and line graph we can analyze that as the file is increasing in a linear the time is also increasing in the same linear fashion

7. Programming Language:

We used C++ programming language to code the Algorithm.

8. Conclusion :

We have succesfully accomplished the project objective to encode any given text and minimize it's size. Although our project has minor limitations that sometimes may provide wrong outputs, say, the file size may not minimize depending on the number of characters provided. We currently developed this to take text inputs and are working towards accepting inputs as a image or a video.

9. References :

1. https://en.wikipedia.org/wiki/Lossless_compression
2. https://en.cppreference.com/w/cpp/chrono/high_resolution_clock/now
3. The Data Compression Book 2nd edition By Mark Nelson and Jean-loup Gailly

10. Appendix :

https://github.com/Nitheesh3009/CSCE5150_project