HUFFMAN DATA COMPRESSION ALGORITHM

INTRODUCTION:

Data compression involves encoding the given information and representing them using fewer bits than the original representation. A simple characterization of data compression is that it involves transforming a string of characters in some representation (such as ASCII) into a new string (of bits, for example) which contains the same information but whose length is as small as possible. It is useful because it reduces the resources required to store and transmit data. As a tradeoff for cheaper and faster transmission of data, computational resources are consumed in the compression and the decompression process.

Data compression paradigm involves trade-off among many factors, like Compression speed, Degree of compression, Decompression speed, and the computational resources required for compressing and decompressing the data.

Data compression can be of 2 types:

1. **Lossless-** All of the information is completely restored after the file is uncompressed. Used for text or spreadsheet files, where losing words or financial data could pose a problem. The Graphics Interchange File (GIF) is an image format used on the Web that provides lossless compression.
2. **Lossy-** When the file is uncompressed, only a part of the original information is still there (although the user may not notice it).Used for video and sound, where a certain amount of information loss will not be detected by most users. The JPEG image file, commonly used for photographs and other complex still images on the Web, is an image that has lossy compression.

In this project I have focussed on **Text Compression** ,which is a lossless data compression. There are three major lossless data compression algorithms **LZW** , **LZSS**, and **HUFFMAN CODING**. HUFFMAN CODING is the latest one among these and mostly used.

OBJECTIVES –

1. To study and analyze the HUFFMAN data compression algorithm.

2. Compression and decompression of a text file using HUFFMAN algorithm.

ALGORITHM ANALYSIS:

Key Points:

1. No 2 characters should have a common prefix.
2. Length of encoded string allocated to a character is inversely proportional to its frequency.

Steps:

Let 'n' be the number of characters in the text and 'd' be the number of distinct characters in the text and 'T' be the total number of characters in Unicode system.

a) ENCODING:

I. Reading the file and making a frequency array to store the frequency of characters appearing in the text. O(n)

```
public void Read_InputFromTextFile() throws IOException{
        BufferedReader br=new BufferedReader(new FileReader(file));
        int ch;
        while((ch=br.read())!=-1){
         freq[ch]++;
        }
    }
```

II. Building **Huffman Tree**. O(2 * d log d + T)

Logic –

In **HUFFMAN TREE** each leaf node stores one of the characters present in the input file. The edges of the tree have a label namely ‘0’ and ‘1’ . Each input character is encoded with the labels of the paths used to reach form the root to the leaf containing these character. Here the most frequent letter is placed closest to the root. So the code given to this character is as small as possible.

Steps –

1. Defining a node class having 4 parameters
    - Character
    - Frequency of Character
    - Reference to the left child
    - Reference to the right child

    ```
    public static class Node{
          Node left,right;
          char value='\0';//default value
          int freq;
          Node(char ch,int count){value=ch;freq=count;}
          Node(int count,Node L,Node R){freq=count;left=L;right=R;}
      }
    ```

2. Add all the root nodes in the PriorityQueue sorted in increasing order of character frequencies.

    ```
    for(int i=0;i<BlockSize;i++){
                if(freq[i]!=0)HuffmanTree.add(new Node((char)i,freq[i]));
              }
    ```

3. Pop 2 nodes out of priority queue. Merge them into a new node having frequency equal to sum of frequency of 2 nodes and left child as the first node and right child as the second node.Do this uptil size of queue is equal to 1.The last node is the root of Huffman Tree.

    ```
    while(HuffmanTree.size()!=1){
              //pop 2 least frequent nodes
              Node N1=HuffmanTree.poll(),N2=HuffmanTree.poll();
    ```

```
          //add a new node with the frequency=sum of frequency of 2 nodes
and node N1 as left child and N2 as right child
          HuffmanTree.add(new Node(N1.freq+N2.freq,N1,N2));
         }
         //the node with the maximum frequency is the root node
         root=HuffmanTree.poll();
```

III. Getting codes for characters from Huffman Tree. O(d^2)

```
//getting codes for different characters from HuffmanTree
        public void GetCode(){
         MakeHuffmanTree HuffmanTree=new MakeHuffmanTree(freq);

         GetCode(HuffmanTree.root,new StringBuilder());
        }
        //dfs traversal of Huffman Tree
        public void GetCode(Node N,StringBuilder sb){
         //root node
         if(N.left==null&&N.right==null){
          code[N.value]=sb.toString();
          TotalBits+=freq[N.value]*code[N.value].length();
          return;
         }
         if(N.left!=null){
          GetCode(N.left,sb.append((char)0));
          sb.deleteCharAt(sb.length()-1);
         }
         if(N.right!=null){
          GetCode(N.right,sb.append((char)1));
          sb.deleteCharAt(sb.length()-1);
         }
        }
```

IV. Encoding the text with the codes given by Huffman Tree. O(n * d)

```
//encoding the input file
        public void Encode_InputFromTextFile()throws IOException{
         BufferedReader br=new BufferedReader(new FileReader(file));
        PrintWriter pw=new PrintWriter(new FileWriter(tofile));
        int bitcount=0;
        char write='\0';
           int ch;boolean start=true;
           while((ch=br.read())!=-1){
            for(int i=0;i<code[ch].length();i++){
             char c=code[ch].charAt(i);
             if(TotalBits<BitLength){
              pw.print(c);continue;
             }
             if(start){
              write|=c;start=false;
             }
             else{
              write<<=1;
              write|=c;
             }
             bitcount++;
             if(bitcount==BitLength){
              pw.print(write);
              write='\0';
              start=true;
              TotalBits-=BitLength;
              bitcount=0;
             }
            }
           }
           pw.flush();pw.close();
       }
```

V. Writing frequency array to text file to be read during decoding. O(T)

```
//writing frequency array to text file
       public void WriteFrequencyArray()throws IOException{
        PrintWriter pw=new PrintWriter(new FileWriter(freqfile));
        for(int i=0;i<BlockSize;i++){
         if(freq[i]!=0)pw.println(i+" "+freq[i]);
```

```
    }pw.print(TotalBits);
    pw.flush();pw.close();
   }
```

b) DECODING:

I. Reading the frequency text file . O(d)

```
//reading frequency array from file
     public void ReadFrequencyArray()throws IOException{
      BufferedReader br=new BufferedReader(new FileReader (freqfile)
);
      String s[];
      while((s=br.readLine().trim().split("\\s+")).length!=1){
       freq[Integer.parseInt(s[0])]=Integer.parseInt(s[1]);
      }
       TotalBits=Integer.parseInt(s[0]);
     }
```

II. Building Huffman Tree O(2 * d log d + T)

III. Getting codes for characters from Huffman Tree. O(d^2)

Steps II. And III. similar to encoding.

IV. Decoding the encoded file O(n * d)

```
//Decoding the Encoded File
     public void Decode_EncodedFile()throws IOException{
      BufferedReader br=new BufferedReader(new
FileReader(encode));
      PrintWriter pw=new PrintWriter(new FileWriter(decode));
      int ch;
      Node node=root;
      while((ch=br.read())!=-1){
       if(TotalBits<BitLength){
        if((char)ch==0){
```

```
  node=node.left;
 }else{
  node=node.right;
 }
 if(node.value!='\0'){
  pw.print((char)node.value);
  node=root;
 }
 continue;
}
for(int i=(BitLength-1);i>=0;i--){
 int c=(ch>>i)&1;
 if(c==0){
  node=node.left;
 }else{
  node=node.right;
 }
 if(node.value!='\0'){
   pw.print((char)node.value);
   node=root;
  }
 }TotalBits-=BitLength;
}
pw.flush();pw.close();
}
```

IMPLEMENTATION AND RESULTS:

Text files with varying sizes were compressed using the algorithm.

Data compression ratio = **Uncompressed File / Compressed File**

`Name of file	Original Size(kb)	Time Taken to compress(sec)	Compressed File Size(kb)	Compression Ratio	Time Taken to decompress(sec)
small.txt	**15**	**0.189426752**	**14**	**1.07143**	**0.05273743**

medium.txt	**94**	**0.227723676**	**80**	**1.175**	**0.097320355**
large.txt	**163**	**0.291876388**	**142**	**1.148**	**0.105364526**

References:

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