School of Computer Science
University of Petroleum & Energy Studies, Dehradun

Mid Term Report (2018-19)

PROJECT TITLE

Lossless Data Compression Algorithms and their Comparison.

ABSTRACT

In today's world, With the advent of the Internet and mobile devices with limited resources and with the growing requirements of information storage and data transfer, Cloud Computing has become an important aspect, but cloud computing also require physical infrastructure, somewhere down the lane. This exponential sub purge of data leads to high demand for data processing that leads to a high computational requirement which is usually not available at the user's end. Compression reduces the redundancy in data representation thus increasing effective data density. [1] Data Compression is a technique which is used to decrease the size of data. This is very useful when some huge files have to be transferred over networks or being stored on a data storage device and the size is more than the capacity of the data storage or would consume so much bandwidth for transmission in a network. With the limited physical infrastructure for storage, data compression has gained even more importance these days. There are number of data compression algorithms, which are dedicated to compressing different data formats. Even for a single data type, there are number of different compression algorithms, which use different approaches. In this project, we will examine lossless data compression algorithms like Huffman encoding algorithm, Lempel-Ziv-Welch algorithm, and Shannon-Fano algorithm and comparing their performance. [2]

Keywords: Cloud Computing, Data Compression, Huffman encoding algorithm, Lempel-Ziv-Welch algorithm, Shannon-Fano algorithm.

INTRODUCTION

Compression is the art of representing the information in a compact form rather than its original or in uncompressed form [3]. In other words, using the data compression, the size of a particular file can be reduced. This is very useful when processing, storing or transferring a huge file, which needs lots of resources. If the algorithms used to encrypt works properly, there should be a significant difference between the original file and the compressed file. When data compression is used in a data transmission application, speed is the primary goal. Speed of transmission depends upon the number of bits sent, the time required for the encoder to generate the coded message and the time required for the decoder to recover the original message. In a data storage application, the degree of compression is the primary concern. There are two types of data compressions ie. Lossless data compression and Lossy data compression.

Table 1.0 :Comparison between Lossy and Lossless Data Compression Technique.

Lossless data compression	Lossy data compression
In Lossless data compression algorithms, the original data can be recovered from compressed data after applying decompression algorithm	In Lossy compression algorithms it permanently reduces the original data by eliminating certain information, especially redundant information, after decompressing the file only the part of original data is recovered.
Lossless compression is generally used for text data or spreadsheet files, where even a very small amount of data loss can be detected by users.	Lossy compression is generally used for video and sound, where a certain amount of information loss will not be detected by most users.
No loss in information so compression rate is small.	In return for accepting this distortion in reconstructed data we obtain high compression rate.
Less data can be accommodated in channel.	More data can be accommodated in channel.
E.g(*i*)*Fax Machine*,(*ii*)*Radiological Imaging*	*E.g*.(*i*)*Telephone System*,(*ii*)*Video CD*

Various lossless data compression algorithms have been proposed and used. Some of the main techniques in use are the Huffman Coding, Run Length Encoding, Arithmetic

Encoding and Dictionary Based Encoding [4]. We will examine the performance of the Huffman Encoding Algorithm, Shannon Fano Algorithm, and Lempel Zev Welch Algorithm. In particular, performance of these algorithms in compressing text data will be evaluated and compared.

Shannon-Fano Algorithm:

In Shannon–Fano Algorithm, the symbols are arranged in order from most probable to least probable, and then divided into two sets whose total probabilities are as close as possible to being equal. All symbols then have the first digits of their codes assigned; symbols in the first set receive "0" and symbols in the second set receive "1". As long as any sets with more than one member remain, the same process is repeated on those sets, to determine successive digits of their codes. When a set has been reduced to one symbol, of course, this means the symbol's code is complete and will not form the prefix of any other symbol's code. The algorithm works, and it produces fairly efficient variable-length encodings; when the two smaller sets produced by a partitioning are in fact of equal probability, the one bit of information used to distinguish them is used most efficiently. Unfortunately, Shannon–Fano does not always produce optimal prefix codes. [5]

Huffman Encoding:

The Huffman encoding works on variable length encoding rather than fixed length encoding. frequency of every character is calculated, and the lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code. The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream for this task a binary tree is created using the symbols as leaves according to their probabilities and paths of those are taken as the code words.

Following are the steps of algorithm-

1. Count the frequency of each character in a text file to be encoded.
2. Create a node of each different character and store them in the queue in ascending order of their frequency.

3. Build a tree by removing the first two elements of the queue and create a new node by joining those two nodes, keeping the first node on left and second on right and then the weight of new node will be the sum of those two nodes then add the new node formed in the queue.
4. Complete building the tree.
5. Assign 0 to left edge and 1 to right edge at every level.
6. Now we can write the Huffman code for each character using the tree and combination of 0's and 1's.

The Lempel-Ziv Welch Algorithm:

It is a dictionary-based compression algorithm As in Dictionary the set of all possible word of a language is stored similarly in LZW algorithm a dictionary is used to store or index the previously used string patterns In the compression process those index value is used instead of repeating. The dictionary is created dynamically in the compression process and no need to transfer it with the encoded message for decompressing. In the decompression process, the same dictionary is created dynamically. Therefore, this algorithm is an adaptive compression algorithm.

Performance of compression algorithm [6] is based on space efficiency and the time complexity

The compression behaviour of algorithm is dependent on redundancy of symbol in source file therefore it is difficult to measure the performance of compression algorithm. There are some following measurements used to evaluate the performances of compression algorithms.

Compression Ratio –Compression Ratio is the ratio between the size of the compressed file and the size of the source file

Compression ratio= (size of compressed file)/ (size of source file)

Compression Factor-Compression factor is inverse of Compression ratio It tells how much time our file has been compressed

Compression Factor= (size of source file)/ (size of compressed file)

Saving Percentage-It tales the shrinkage of the source file in percentage

Saving Percentage= (size before compression-size after compression)/ (size before compression) %

Time complexity –The time complexity is measured by the number of clocks used to encode or decode the source code. The algorithm that uses less clock cycle to encode or decode is considered more efficient.

Code Efficiency Average code length is the average number of bits required to represent a single code word. If the source and the lengths of the code words are known; the average code length can be calculated using the following equation.

$$\bar{l} = \sum_{j=1}^{n} p_j l_j$$

, where p_j is the occurrence probability of j th symbol of the source message, l_j is the length of the particular code word for that symbol and $L = \{l_1, l_2, ... l_n\}$.

In this project we will be comparing all the three algorithms on the basis of these factors mentioned above.

OBJECTIVES ACHIEVED

1. Pseudo Code of Huffman Coding, LZW and Shannon-Fano Algorithm has been written.
2. Implementation of Huffman Code and Shannon-Fano has completed.

LITERATURE REVIEW

Data Compression is the way that you can use the space on cloud i.e. Server in an optimal way. In this project we will Lossless Data Compression algorithms which can reconstruct the original message exactly from the compressed message Here is the conclusion of some of the reference paper that we review to make our project better and to know more technologies that we can use in our system.

- In the paper [6] by S.R. KODITUWAKKU, Department of Statistics & Computer Science, University of Peradeniya, Sri Lanka, U. S. AMARASINGHE, Postgraduate Institute of Science, University of Peradeniya, Sri Lanka; Among the

available lossless compression algorithms they considered the Run Length Encoding Algorithm, Huffman Encoding, The Shannon Fano Algorithm, Arithmetic Encoding, The Lempel Ziv Welch Algorithm for study. They carried out an experimental comparison of a number of different lossless compression algorithms for text data. On the basis of compression times, decompression times and saving percentages of all the algorithms, they found that the Shannon Fano algorithm can be considered as the most efficient algorithm among the selected ones. The values which they calculated are in the acceptable range and it also shows better results for the larger files.

- In the paper [7] by Laxmi Shaw, Student Member, IEEE, Daleef Rahman, and Aurobinda Routray, Senior Member, IEEE; The authors have examined the different lossless compression methods for single and multichannel EEG signals, and their performance with respect to their relative Compression ratios has been analysed. They evaluate their proposed algorithms, analysis of which showed that a very high CR (Compression Ratio) in different publicly available database. They also analysed that among the existing methods for the single-channel EEG compression scheme, the linear prediction followed by the context-based error modelling showed the best results. The increase in CR by applying the context-based error modelling is high for the first-order predictor, whereas the increase is small for higher-order predictors. The MVAR model and the bivariate autoregression model were examined for the multichannel EEG compression. The results show that these proposed methods in combination outperform the existing MVAR and the bivariate autoregression model.

- In the paper [8] by Mohammad Hosseini, Network Systems Lab; In this research paper author introduced two types of compression, lossless and lossy compression, and some major concepts, algorithms and approaches in data compression and discussed their different applications and the way they work. They also evaluated two of the most important compression algorithms based on simulation results. Then as his next contribution, he thoroughly discussed two major everyday applications regarding data compression; JPEG as an example for image compression and MPEG as an example of video compression in our everyday life. At the end of this survey he discussed major issues in leveraging data compression algorithms and the state-of-the art research works done regarding energy saving in topworld-discussed area in networking which is Wireless Sensor Networks.

RELATED WORK

1) In this research paper[1] they describe the design, analyze the structure, and evaluate the performance of SPDP, an automatically synthesized lossless compression algorithm for single- and double-precision floating-point data. It is the best-compressing out of the 9,400,320 possible four-stage algorithms that can be built from our set of 48 algorithmic components that does not include any bit-level coders. SPDP yields the highest compression ratio on eleven of the 26 tested datasets and outperforms all of the evaluated compressors except Zstd.

   More importantly, their analysis represents a first step in a new direction aimed at improving their understanding of how to build effective domain-specific compression algorithms. First, by systematically generating candidates and analyzing the structure of the best resulting algorithm, they were able to gain insight into its operation and learned how to handle mixed precision datasets. Second, they were able to demonstrate that a competitive algorithm can be created based solely on transformations that do not process data at bit granularity.
2) In this paper author propose simple, effective and very fast compression/decompression methods for FASTQ genomic data using general purpose compression components in a columnar compression model, robust to eventual changes in the FASTQ format specification.The compression speed as observed for ERR317482_1 FASTQ data (≈ 7GB) in our tests is about 40X better than default gzip and 4.5X-7.4X faster than the latest, best proposals like DSRC v2 and slimfastq as featured in [2]. Decompression speeds are 4X-5X faster than gzip and 5X-17X faster than the specialized FASTQ compressors, DSRC and slimfastq- a more robust re-implementation of Fqzcomp/Fastqz. Compression ratios are 5.6%-11.4% better than gzip flavors and just 6.4% - 9.7% less than the best, more complex coders as reported in [2].

3) Data compression is an important technique to improve the performance and space efficiency for flash-based storage systems .In this research paper Author[3] describe the employing fixed compression algorithms, as in most current flash-based storage products that incorporate data compression, fails to recognize and exploit the significant diversity in compressibility and access patterns of data and misses the opportunity to improve system performance, space efficiency or both. EDC is proposed in this paper to exploit the compression diversity of the workload characteristics .More specifically, for compressible data blocks EDC employs algorithms with higher compression ratios in time periods with lower system utilization and algorithms with lower compression ratios in time periods with higher system utilization. For non-compressible (or very lowly compressible) data blocks, it will write them through to the flash storage directly without any compression. Their extensive trace-driven evaluations on a lightweight implementation of the EDC prototype show that EDC achieves a much better trade-off between performance and space efficiency than the state-of-the-art schemes with fixed algorithms.

PROBLEM STATEMENT

Storage on the cloud is a limited resource. Even though more storage space can be purchased, it seems better to utilize the given space to the fullest. The solution to this problem is data compression. Compress data to save space and then store it on the cloud, also in doing so, we save the data transmission cost over the network and make our cloud storage even more efficient.

SYSTEM REQUIREMENTS

- **Hardware Interface:**
  - 64 bits processor architecture supported by windows.
  - Minimum RAM requirement for proper functioning is 8 GB.
  - Required input as well as output devices.

- **Software Interface:**
  - C Compiler (GCC).
  - AWS CLOUD SERVICES.
  - Socket Programming Library in C.

IMPLEMENTATION

Data Compression will be done by these of three Algorithms defined below:

Huffman Coding:

Huffman coding is lossless data compression algorithm that comes under Greedy Algorithm. The basic idea behind Algorithm is to assign variable length code to each character based on their frequency
There are two part of Huffman encoding
a) Creating a Huffman tree
b) Traverse the tree and assign code to each character

a. Creating a Huffman tree

Step1- Create a leaf node for each unique character and build a min heap for each character
Step2- Extract two node (Whose frequency is minimal) from min heap say node1 and node2

Step3- Create a new internal node whose frequency is equal to sum of two extracted node assign node 1 as a left child and node 2 as aright child
Step4-Repeat step2 and step3 until the min heap does not contain only one node. The remaining node is root node and the tree has been completed

b. Traverse the tree and assign code to each character

Step1- Traverse the tree formed starting from the root. Maintain an auxiliary array.
Step2- While moving to the left child, write 0 to the array.
Step3- While moving to the right child, write 1 to the array.
Step4- Print the array when a leaf node is encountered.

Pseudo code for Huffman encoding:

Let C is set of n character such that every character c belongs to C every character has a attribute is frequency. And Q is a min priority queue based on frequency of characters

```
Huffman(C)
{
// input : C set of n character with their frequency
// Output : Huffman tree
  n=|C|
  Q=C
  for i=1 to n-1 do
  Allocate a new internal node z
  z.left=x=Extract-Min(Q)          //Extracting two internal nodes with lowest frequency
  z.right=y=Extract-Min(Q)         //And assigning that sum to new internal node
  z.freq=x.freq+y.freq
  Insert(Q,z)
  return Extract-Min(Q)  //return the root of the tree
}
```

Time complexity of creating Huffman tree

= (time complexity to Extract min node)*2(n-1) // where n is number of character
=O(logn)*2n
=O(nlogn)

Shannon-Fano : Shannon-fano algorithm is a lossless data compression algorithm named after Claude Shannon and Robert Fano, in this technique we constrtuct a code for each character which depends upon the frequency of that character and then encode the file accordingly.
There are two part of Huffman encoding
a) Creating a Shannon-Fano tree
b) Traverse the tree and assign code to each character.

Creating a Shannon-Fano tree

Step 1: Calculatee frequency of each unique character.
Step 2: Calculate probability of each unique character in the file.
Step 3: Arrange the probability in decreasing order.

Step 4: Divide the Array in two parts in such a way that sum of probablity of one part almost equal to sum of probabilty of other part.

Step 5: Keep diving the array till only two probabilty are left in sub-parts.

Step 6: Assign 0 to all left sub-edges and 1 to all right sub-edges .

Traverse the tree and assign code to each character.

Step 1:Calculate the code for each character by taking 0 and 1 values of the edges required to reach the leaf node of that character.

Step 2:Traverse the file and write corresponding code of each character in a binary file.

Step 3:Save Binary file.

Pseudo code for Shannon-Fano encoding:

Take two Arrays Probabilty[] which consist probabilty of each character ,and Character[] that consist the consist of all the character in the same order of there probabilty.

Example:

```
Probability[]={0.30,0.25,0.15,0.12,0.10,0.08}
Character[]={'a','c','s','b','m','n'}
Shannonfano( initial, final, code)
{
 If initial=final:
   print code and return.
 Else
       Assign first=initial,last=final
       Assign sumFront=probability[first], sumLast=probability[last]
       while first!=last:
             If sumFront>sumLast:
                Decrement last by 1
                Assign sumLast=sumLast+probability[last]
             Else
                Increment first by 1
                Assign sumFront=sumFront+probability[first]
        Assign code=code*10
       Shannonfano(initial,first,code+0)
       Shannonfano(first+1,final,code+1)
}
```

Time-Complexity of Shannon-Fano algorithm

$T(n) = T(n/2) + T(n/2) + n$

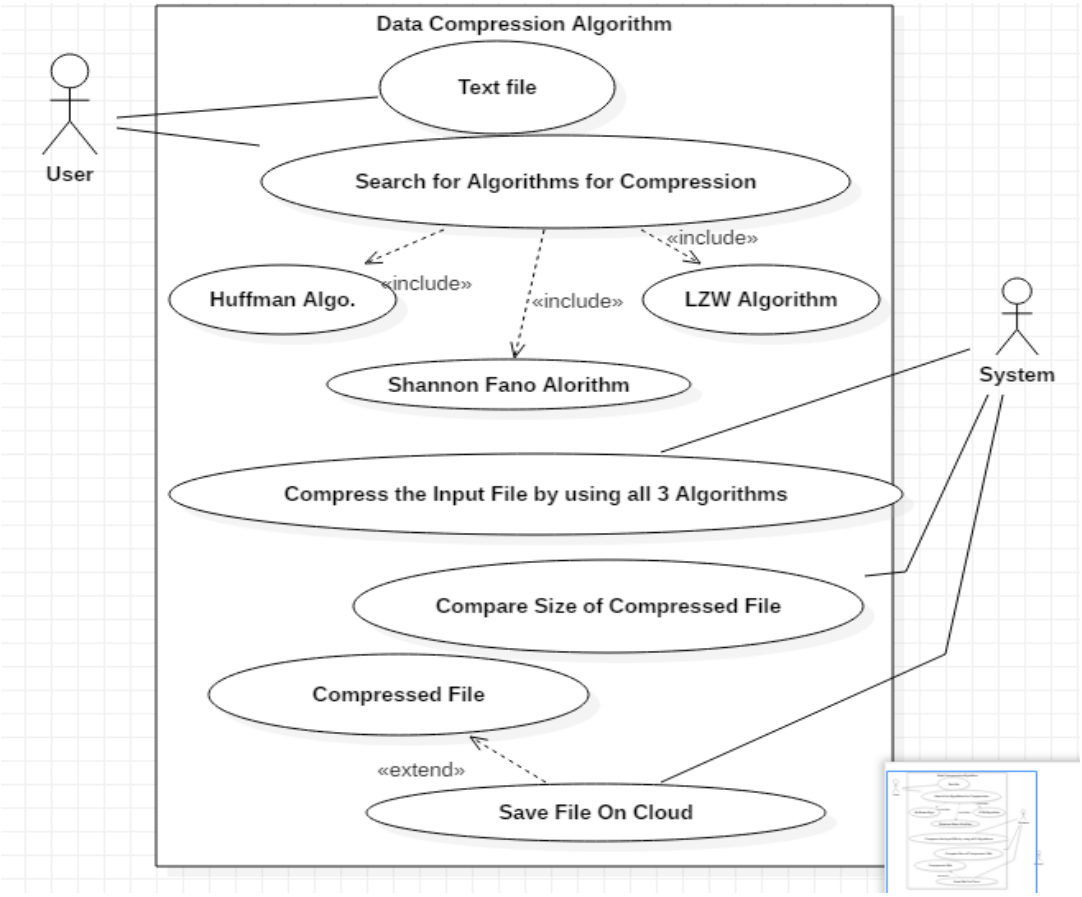
$T(n) = 2T(n/2) + n$

$=O(nLog_2(n))$

Lempel-Ziv-Welch :

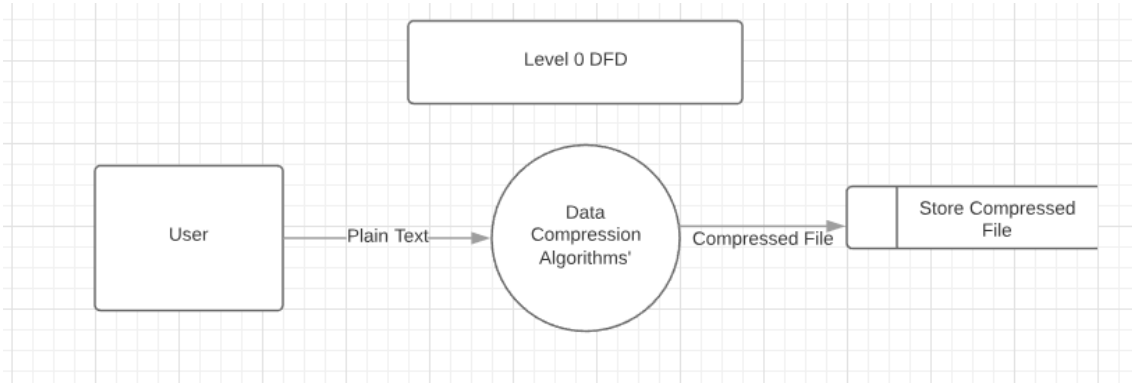
DESIGN

USE CASE:

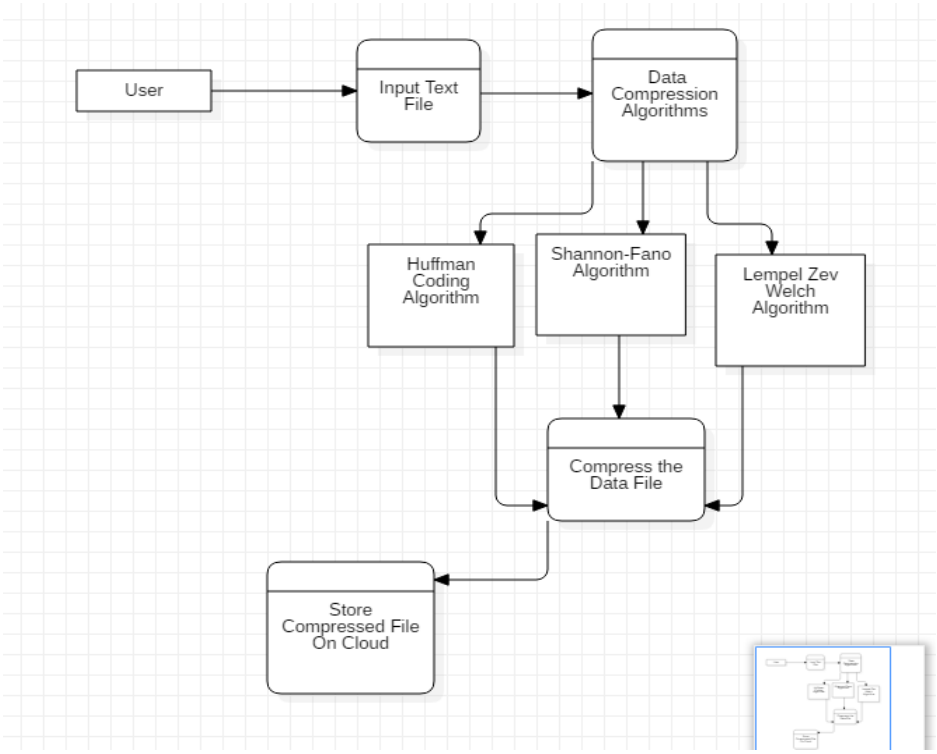


DFD:

1. **Level-0 DFD**



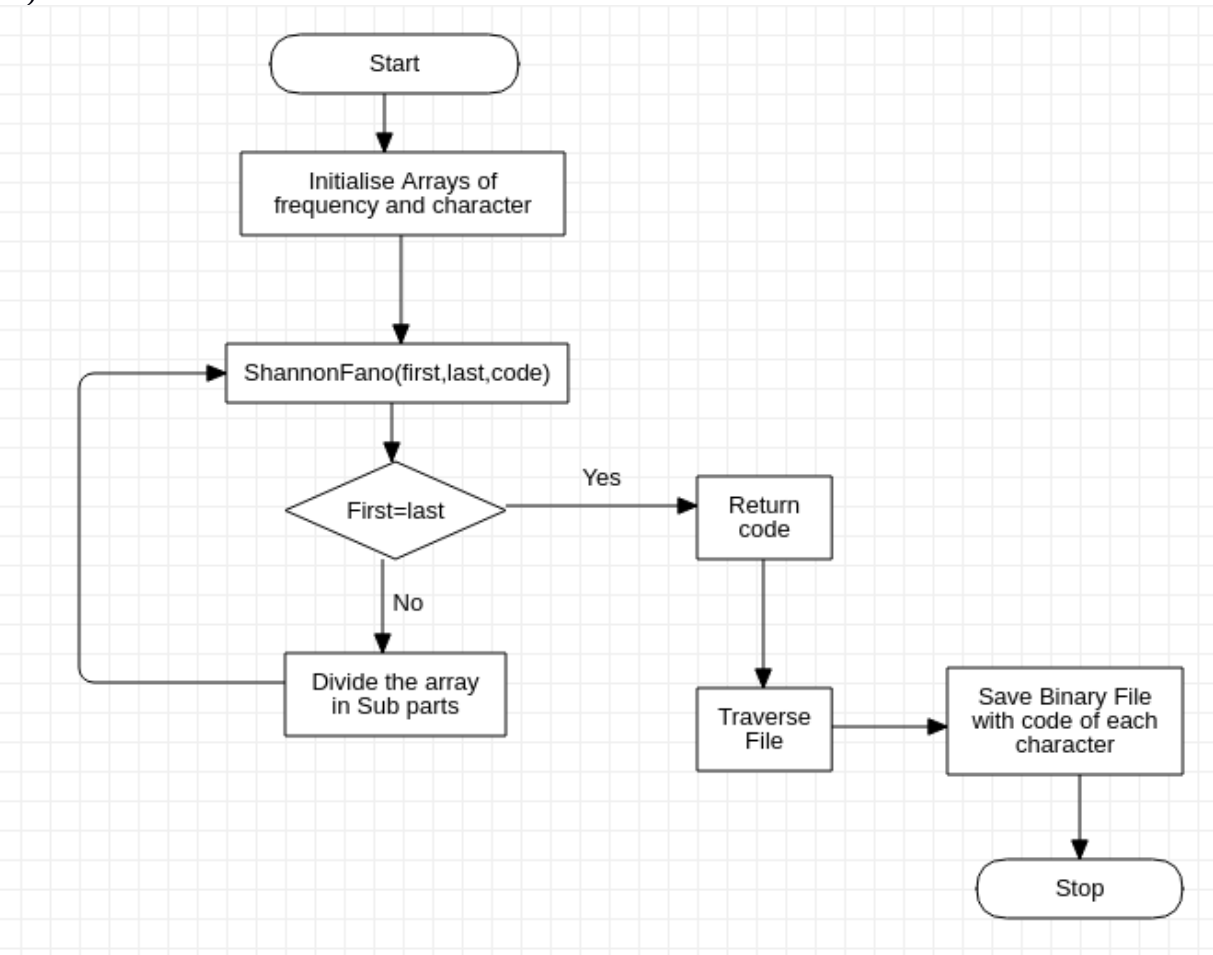
2. **Level-1 DFD**



FLOW CHART:

1) **HUFFMAN CODING**

2) **SHANNON-FANO**



3) **Lempel Zev Welch**

REFERENCE

[1] Monika Soni , Dr Neeraj Shukla “Data Compression Techniques in Cloud Computing”

[2] Mohammad Hosseini “A Survey of Data Compression Algorithms and their Applications”

[3] Pu, I.M., 2006, Fundamental Data Compression, Elsevier, Britain.

[4] Kesheng, W., J. Otoo and S. Arie, 2006. Optimizing bitmap indices with efficient compression, ACM Trans. Database Systems, 31: 1-38.

[5] https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=15&ved=2ahUKEwjW8piM_q_dAhVHyrwKHXE2DHoQFjAOegQIABAC&url=http%3A%2F%2Fecehithaldia.in%2Fteaching_material%2FShanon-Fano1586521731.pdf&usg=AOvVaw0MHM4foSS-sDhzqyRAVfaE

[6] S.R. Kodituwakku ,U. S. Amarasinghe “Comparision of Lossless data compression algorithms for text data”

[7] Highly Efficient Compression Algorithms for Multichannel EEG,Laxmi Shaw , Student Member, IEEE , Daleef Rahman, and Aurobinda Routray, Senior Member, IEEE

[8] A Survey of Data Compression Algorithms and their Applications ,Mohammad Hosseini

[9] Network Systems Lab, School of Computing Science, Simon Fraser University, BC, Canada,Email: mohammad hosseini@sfu.ca

A APPENDIX I PROJECT CODE

1.Huffman Encoding

```
#include <stdio.h>
#include <stdlib.h>
#define HEIGHT 100
int frequency[128]={0};
int temp[128];

struct NODE {
   char data;
   int freq;
   struct MinHeapNode *left, *right;
};
typedef struct NODE Node;
typedef struct  {
   int size;
   int capacity;
   Node** array;
}MinHeap;
Node* newNode(char data, int freq)
{
   Node* temp= (Node*)malloc
(sizeof(Node));

   temp->left = temp->right = NULL;
   temp->data = data;
   temp->freq = freq;

   return temp;
}
void sort(int *frequency ,int *temp)
{
  int a,i,j;
  for (i = 0; i < 128; ++i) {
    if(!(frequency[i]>0.0))
      {
       continue;
      }
  for (j = i + 1; j < 128; ++j)
    {
      if (frequency[i] < frequency[j]) {
      a =  frequency[i];
```

```
    frequency[i] = frequency[j];
    frequency[j] = a;
    a=temp[i];
    temp[i]=temp[j];
    temp[j]=a;
    }}}}
void file_read()
{
    FILE *fp;
    int length=0;
    int i;

    printf( "Opening the file test.c in read mode\n" ) ;
    fp = fopen( "test.c", "r" ) ;
    if ( fp == NULL ){
      printf( "Could not open file test.c\n" ) ;
      return 1;
    }
    printf( "Reading the file test.c\n" ) ;
    int c;
    while((c = fgetc(fp))!=EOF){
      frequency[c]++;
      length++;
    }
    for(i=0;i<128;i++)
    {
       temp[i]=i;
    }
    sort(frequency,temp);
    for(i=0;i<128;i++)
    {
      if(frequency[i]>0)
      printf("%c is %d times\n",temp[i],frequency[i]);
    }
    printf("\nClosing the file test.c\n") ;
    fclose(fp);
}

MinHeap* createMinHeap(int capacity)
{
MinHeap* minHeap = (MinHeap*)malloc(sizeof(MinHeap));
   minHeap->size = 0;
   minHeap->capacity = capacity;
   minHeap->array
      = (Node**)malloc(minHeap->
capacity * sizeof(Node*));
   return minHeap;
}

void swapNode(Node** a,Node** b)
{
Node* t = *a;
*a = *b;
*b = t;
}

void minHeapify(MinHeap* minHeap, int idx)
{
int smallest = idx;
```

```
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;

    if (left < minHeap->size && minHeap->array[left]->
freq < minHeap->array[smallest]->freq)
        smallest = left;
if (right < minHeap->size && minHeap->array[right]->
freq < minHeap->array[smallest]->freq)
        smallest = right;
if (smallest != idx) {
        swapNode(&minHeap->array[smallest],
                    &minHeap->array[idx]);
        minHeapify(minHeap, smallest);
    }
}

int isSizeOne(MinHeap* minHeap)
{
return (minHeap->size == 1);
}

Node* extractMin(MinHeap* minHeap)
{
Node *temp = minHeap->array[0];
    minHeap->array[0]= minHeap->array[minHeap->size - 1];

    --minHeap->size;
    minHeapify(minHeap, 0);

    return temp;
}

void insertMinHeap(MinHeap* minHeap,Node* minHeapNode)

{

    ++minHeap->size;
    int i = minHeap->size - 1;

    while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) {

        minHeap->array[i] = minHeap->array[(i - 1) / 2];
        i = (i - 1) / 2;
    }

    minHeap->array[i] = minHeapNode;
}
void buildMinHeap(MinHeap* minHeap)
{

    int n = minHeap->size - 1;
    int i;

    for (i = (n - 1) / 2; i >= 0; --i)
        minHeapify(minHeap, i);
}

void printArr(int arr[], int n)
{
```

```
    int i;
    for (i = 0; i < n; ++i)
        printf("%d", arr[i]);

    printf("\n");
}
int isLeaf(Node* root)

{

    return !(root->left) && !(root->right);
}

MinHeap* createAndBuildMinHeap(char data[], int freq[], int size)

{
   int i;
   MinHeap* minHeap = createMinHeap(size);

    for ( i = 0; i < size; ++i)
        minHeap->array[i] = newNode(data[i], freq[i]);

    minHeap->size = size;
    buildMinHeap(minHeap);

    return minHeap;
}
Node* buildHuffmanTree(char data[], int freq[], int size)

{
    Node *left, *right, *top;
    MinHeap* minHeap = createAndBuildMinHeap(data, freq, size);

    while (!isSizeOne(minHeap)) {
        left = extractMin(minHeap);
        right = extractMin(minHeap);
        top = newNode('$', left->freq + right->freq);
        top->left = left;
        top->right = right;

        insertMinHeap(minHeap, top);
    }

    return extractMin(minHeap);
}
void printCodes(Node* root, int arr[], int top)

{

    if (root->left) {

        arr[top] = 0;
        printCodes(root->left, arr, top + 1);
    }

    if (root->right) {

        arr[top] = 1;
        printCodes(root->right, arr, top + 1);
```

```
    }
    if (isLeaf(root)) {

        printf("%c: ", root->data);
        printArr(arr, top);
    }
}

void HuffmanCodes(char data[], int freq[], int size)
{
    Node* root
        = buildHuffmanTree(data, freq, size);
    int arr[HEIGHT], top = 0;

    printCodes(root, arr, top);
}
int main()
{
int i=0 ,j=0,k=0;
char arr[128];
int freq[128];
file_read();
 for(i=0;i<128;i++)
{
if(frequency[i]>0)
{
freq[k]=frequency[i];
arr[k]=(char)temp[i];
k++;
}
}
int size =k;
HuffmanCodes(arr, freq, size);
return 0;
}
```

test.c contains data:

```
aaabbbccddeeaaaf
```

OUTPUT:

2.Shannon-Fano:

Code:

```
#include<string.h>
#include<stdio.h>
int codeword[128]={0};
void sort(int *frequency ,int *temp)
{
  int a,i,j;
  for (i = 0; i < 128; ++i) {
    for (j = i + 1; j < 128; ++j)
    {
      if (frequency[i] < frequency[j])
      {
        a =  frequency[i];
        frequency[i] = frequency[j];
        frequency[j] = a;
        a=temp[i];
        temp[i]=temp[j];
```

```
        temp[j]=a;
      }
    }
  }
}
void shannonfano(int initial,int final,int code,float *probability,int *character)
{
  if(initial==final)
  {

    codeword[initial]=code;
    return ;
  }
  int first=initial;
  int last=final;
  float sumFront=probability[first];
  float sumLast=probability[last];
  while(first!=last)
  {
    if(sumFront>=sumLast)
    {
      last--;
      sumLast=sumLast+probability[last];
    }
    else{
      first++;
      sumFront=sumFront+probability[first];
    }
  }
  code=code*10;
  shannonfano(initial,first,code+0,probability,character);
  shannonfano(first+1,final,code+1,probability,character);

}
int main(){
      FILE *fp;
      int length=0;
      int frequency[128]={0};
      int i;
      int temp[128];
      printf( "Opening the file test.c in read mode\n" ) ;
      fp = fopen( "test.c", "r" ) ;
      if ( fp == NULL ){
        printf( "Could not open file test.c\n" ) ;
        return 1;
      }
      printf( "Reading the file test.c\n" ) ;
      int c;
      while((c = fgetc(fp))!=EOF){
        frequency[c]++;
        length++;
      }
      for(i=0;i<128;i++)
      {
          temp[i]=i;
      }
      sort(frequency,temp);
      float probab[128]={0.0};
      int last=0;
```

```
    for(i=0;i<128;i++)
    {
       if(frequency[i]==0)
       {
         last=i;
         break;
       }
       probab[i]=frequency[i]/(float)length;
    }
    shannonfano(0,last-1,0,probab,temp);
    for (int i = 0; i < last; ++i)
    {
      printf("code=%d for character=%c\n",codeword[i],temp[i] );
    }
    fclose(fp);
    printf("\nClosing the file test.c\n") ;
    //starting to write in binary file
    FILE *fptr;
     if ((fptr = fopen("output.txt","w")) == NULL){
         printf("Error! opening file");
         // Program exits if the file pointer returns NULL.
         return 0;
     }
    fp = fopen( "test.c", "r" ) ;
    if ( fp == NULL ){
      printf( "Could not open file test.c\n" ) ;
      return 1;
    }
    printf( "Reading the file test.c\n" ) ;
    while((c = fgetc(fp))!=EOF){
      for (int i = 0; i <last; ++i)
      {
        if(temp[i]==c)
        {
          fprintf(fptr,"%d",codeword[i]);
        }
      }

    }
    fclose(fp);
    fclose(fptr);
    return 0;
}
```