**Information Theory and Coding**

**Computer Science and Engineering (CSE)**

B. Tech IVth year 7th semester

**PROJECT REPORT**

ON

**“IMPLEMENTATION OF**

**SHANNON-FANO CODING”**

Submitted by:

Partha Pritam Paul 14-15-059

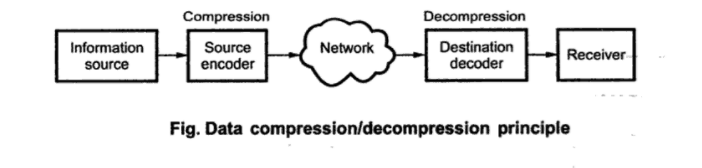
Udit Narayan Bora 14-15-062

Under the Guidance of

Mr. Umakanta Majhi

**ABSTRACT**

In digital communication while transmit the data it is well desire that the transmitting data bits should be as minimum as possible, so to compress the data there are several techniques used. Data compression is one of the most important areas of computer science. A popular living example is ZIP file format, which is widely used in PC. It not only provides compress functions, but also offers archive tools (Archiver) that can store many files in a same root file. In this paper we have implemented a Shannon-Fano algorithm for data compression through MATLAB.



The Shannon-Fano technique was proposed in Shannon's "A Mathematical Theory of Communication", his 1948 article, introducing the field of information theory. The method was attributed to Fano, who later published it as a technical report. Shannon–Fano coding should not be confused with Shannon coding, the coding method used to prove Shannon's noiseless coding theorem, or with Shannon–Fano–Elias coding (also known as Elias coding), the precursor to arithmetic coding.

**INTRODUCTION**

In today's world of digitalization, all the data which is to be processed or transmit or received that should contain memory or bits as minimum as possible. So to reduce the bits or to compress the data, there are several techniques, one of which technique is called Shannon-Fano algorithm, named after **Claude Shannon** and **Robert Fano**, is a technique for constructing a “prefix code” based on a set of symbols and their probabilities (estimated or measured). It is suboptimal in the sense that it does not achieve the lowest possible expected code word length like Huffman coding; it does guarantee that all codeword lengths are within one bit of their theoretical ideal *I(x) = − log P(x).*

The algorithm 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; the set of probabilities {0.35, 0.17, 0.17, 0.16, 0.15} is an example of one that will be assigned non-optimal codes by Shannon–Fano coding.

For this reason, Shannon–Fano is almost never used; Huffman coding is almost as computationally simple and produces prefix codes that always achieve the lowest expected code word length. Shannon–Fano coding is used in the IMPLODE compression method, which is part of the ZIP file format, where it is desired to apply a simple algorithm with high performance and minimum requirements for programming.

**SHANNON-FANO ALGORITHM:**

*.* A Shannon-Fano tree is built according to a specification, designed to define an effective code table. The actual algorithm is simple. For a given list of symbols, develop a corresponding list of probabilities or frequency counts so that each symbol’s relative frequency of occurrence is known.

Shannon-Fano source encoding follows the steps:

1. Order symbols mi in descending order of probability
2. Divide symbols into subgroups such that the subgroup’s probabilities (i.e.

information contests) are as close as possible.

1. Allocating codewords: assign bit 0 to top subgroup and bit 1 to bottom subgroup.

This means that the codes for the symbols, in the first part will all start with 0, and the codes in the second part will all start with 1.

1. Iterate steps 2 and 3 as long as there is more than one symbol in any subgroup,

subdividing groups and adding bits to the codes until each symbol has become a corresponding code leaf on the tree.

1. Extract variable-length codewords from the resulting tree (top-down). Codewords must follow Prefix condition i.e. no codeword forms a prefix for any other codeword, so they

can be decoded unambiguously.

**Computing Avg. Code Length and Entropy**

Let, p­­i =Probability of symbols, and

li=Length of codeword,

m=Number of symbols

then,

Avg. code length(Lavg)=∑(li)(pi)

Entropy(H)= -∑pi log(pi) ;log is of base 2

Efficiency(η)= H/Lavg

Redundancy= 1-Efficiency

**Shannon-Fano Coding Example**

* **Example for 8 symbols:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Symbol | A | B | C | D | E | F | G | H |
| Probability | 0.20 | 0.27 | 0.06 | 0.16 | 0.17 | 0.04 | 0.04 | 0.06 |

**Solution:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Symbol** | **Probability** | **Step 1** | **Step 2** | **Step 3** | **Step 4** | **Codeword** |
| B | 0.27 | 0 | 0 |  |  | 00 |
| A | 0.20 | 0 | 1 |  |  | 01 |
| E | 0.17 | 1 | 0 | 0 |  | 100 |
| D | 0.16 | 1 | 0 | 1 |  | 101 |
| C | 0.06 | 1 | 1 | 0 | 0 | 1100 |
| **Symbol** | **Probability** | **Step 1** | **Step 2** | **Step 3** | **Step 4** | **Codeword** |
| H | 0.06 | 1 | 1 | 0 | 1 | 1101 |
| F | 0.04 | 1 | 1 | 1 | 0 | 1110 |
| G | 0.04 | 1 | 1 | 1 | 1 | 1111 |

The Entropy of the Source is:

H= -∑ pi log(pi)

= -(0.27\*log2(0.27) + 0.20\*log2(0.20) + 0.17\*log2(0.17) +

0.16\*log2(0.16) + 0.06\*log2(0.06) + 0.06\*log2(0.06) +

0.04\*log2(0.04) + 0.04\*log2(0.04))

=2.6906 bit/symbol

The average length of the binary code is:

Lavg =∑ (li )((pi)

=2(0.27) +2(0.20) +3(0.17) +3(0.16) +4(0.06) +4(0.06) +4(0.04) +4(0.04)

=2.73 bit/symbol

Efficiency:

η= H/Lavg

=2.6906/2.73

=0.9856

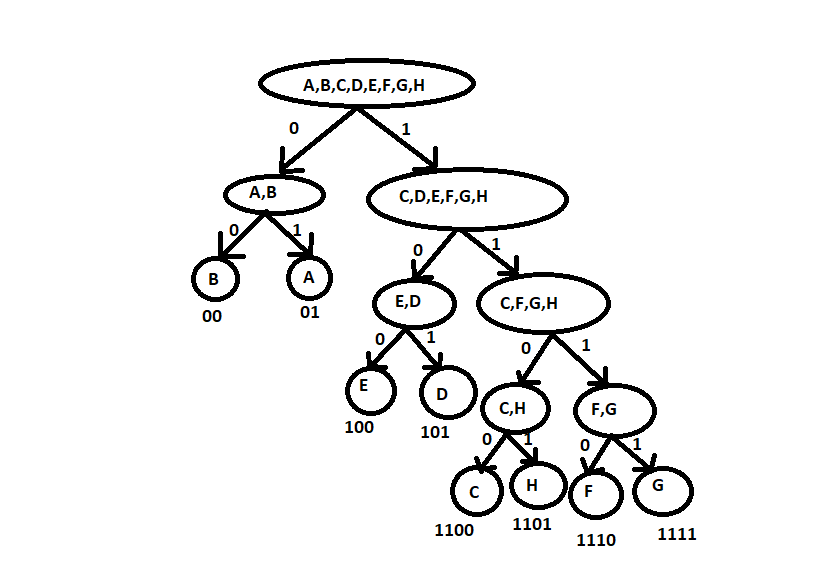
Redundancy

R=1- η

=1-0.9856

=0.0144

**Shannon-Fano Tree:**



**SOURCE CODE:**

#include<stdio.h>

#include<string.h>

struct node {

char sym[10];

float pro;

int s[20];

}s[20];

typedef struct node node;

int g\_level = 0;

void **shannon** (int start, int end, node s[], char code[20][20],int level)

{

int i=start;

int j=end;

float isum = s[i].pro, jsum = s[j].pro;

if(level>g\_level) {

g\_level = level;

}

while(i<(j-1)) {

while(isum>jsum && i<(j-1)){

j--;

jsum += s[j].pro;

}

while(isum<jsum && i<(j-1)) {

i++;

isum += s[i].pro;

}

}

if(i==start) {

code[start][level]='0';

}

else if((i-start)>=1) {

int k;

for(k=start;k<=i;++k)

code[k][level] = '0';

**shannon**(start,i,s,code,level+1);

}

if(j==end) {

code[end][level]='1';

}

else if((end-j)>=1) {

int k;

for(k=j;k<=end;++k)

code[k][level] = '1';

**shannon**(j,end,s,code,level+1);

}

}

int main(int argc, char\* argv[])

{

float a[20], temp, x, total=0;

int n,i,j;

char ch[10];

printf("Enter the number of symbols:");

scanf("%d",&n);

for(i=0;i<n;i++){

printf("Enter symbol %d ---> ",i+1);

scanf("%s",ch);

strcpy(s[i].sym,ch);

}

for(i=0; i<n; ++i)

{

printf("\n\tEnter probability for %s ---> ",s[i].sym);

scanf("%f",&x);

s[i].pro=x;

total=total+s[i].pro;

if(total>1) {

printf("\t\tThis probability is not possible. Enter new probability");

total=total-s[i].pro;

i--;

}

}

if(total<1) {

return 0;

}

// Take the symbols and sort them as user enters them using insertion sort

for(i=1;i<n;i++) {

temp = s[i].pro;

j=i-1;

while(temp>s[j].pro && j>=0) {

s[j+1].pro = s[j].pro;

--j;

}

s[j+1].pro = temp;

}

char code[20][20];

for(i=0; i<n; ++i) {

// Mark row as invalid

for(j=0;j<n;j++) {

code[i][j] = 'X';

}

}

**shannon**(0,n-1,s,code,0);

printf("---------------------------------------------“)

printf("\nSymbol\tProbability\tCode\n");

// Print the data and code

for(i=0; i<n; ++i) {

printf("%S\t %f\t :", s[i].sym,s[i].pro);

for(j=0; j<=g\_level; j++)

{

if(code[i][j]!='X')

printf("%c",code[i][j]);

}

printf("\n");

}

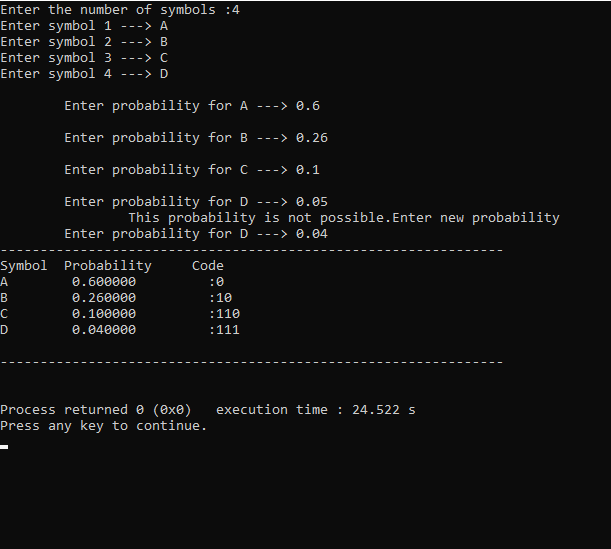
printf("\n---------------------------------------");

printf("\n\n");

return 0;

}

**OUTPUT**

****

**CONCLUSION**

**REFERENCE**