Designing and simulating an algorithm for n-gram Huffman source codes using MATLAB

HASHIM RAWASHDEH
SALEH ALGHOUL
24/5/2021

Abstract

An important result of Shannon's first theorem, or "the noiseless coding theorem" is that the coding efficiency may be improved by coding extensions of the source to values arbitrarily approaching unity for large number of extensions [1]. This report uses n-gram Huffman coding algorithm to show that the coding efficiency does in fact increase as the number of extension or gram increases. A complete system model of the system is presented and simulated using Matlab software.

1.Introduction

Huffman coding is a variable length coding technique for compressing data without losing any information. It is generally useful to compress the data in which there are frequently occurring characters. Huffman coding technique generates shorter binary codes for encoding characters that appear more frequently in the data. The binary codes generated are prefix codes; that is, the bit string representing a particular symbol is never a prefix of the bit string representing any other symbol. For a discrete memoryless information source S described by its characters $\{\boldsymbol{C_1}, \boldsymbol{C_2}, \boldsymbol{C_3} \dots \boldsymbol{C_i}\}$ that occur with probabilities $\{\boldsymbol{P(1)}, \boldsymbol{P(2)}, \boldsymbol{P(3)} \dots \boldsymbol{P(i)}\}$, the entropy of the source **H(S)** is defined as: $H(S) = -\sum_i P(i) \log_2 P(i) \dots (1)$ which is a measure of the information that is acquired by the knowledge of S ,or the information content of S per source output [1]. It is advised to use variable length codes when encoding characters from the source S for transmission over a noiseless channel. Given that the probability of each codeword is P(i) and the length of each codeword is L(i), the average codeword length is defined as $L = \sum_i P(i)L(i) \dots (2)$. To characterize the quality of the n-gram Huffman coding algorithm, coding efficiency $\boldsymbol{\eta}$ is defined as $\eta = H(S)/L \dots (3)$. As motivated, in this work, we model the complete system of generating the source. Then, present an algorithm that encodes the source's characters for n-gram extensions (1,2,3,4,5) using Huffman coding technique as flow charts and implement it using Matlab software. Finally, we discuss the objective (3) presented above.

2. System Model

The described system is considered for a discrete memoryless source of seven characters only. The system is modelled with five subsystems as shown in figure (1) below, each one of the subsystems does a particular function to be discussed separately in a dedicated section. In this system, seven random characters are generated by the source. These characters are assigned probabilities using the

probability assigner. Then, the five source extension's probabilities vectors are generated using the third block. Then the designed algorithm is applied to these vectors. Finally, the coding efficiency for each extension is calculated and compared with each other.

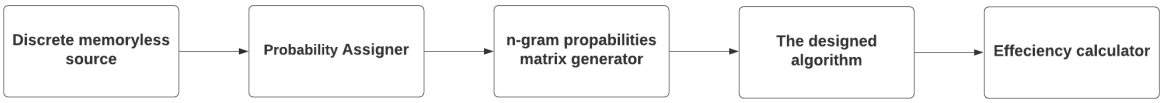


Fig.1. *Block diagram of the system*

2.1 The Discrete Memoryless Source

A discrete memoryless source (DMS) is a discrete source whose current output letter is statistically independent of all past and future outputs [2]. In this section, seven statistically independent random characters are generated. The following three lines of code do so. The first line generates the ASCII codes of the characters mentioned in Appendix A, the second line samples seven random codes from them and the last line converts each ASCII code to its corresponding character.

```
 characters=(char(32:126));
DoupleInputText= randsample(32:126,7);
InputText=char(DoupleInputText);
```

2.2 The Probability Assigner

There are two ways to define probability. The first is based on fundamental axioms in set theory; This definition is mathematically more rigorous than the second but, it is harder to capture a practical sense out of it. Because of that, the second definition which is the ***relative frequency*** definition is defined here and used in Appendix B, which is based more in engineering and common sense. The relative frequency definition of probability is as follows, if an experiment is repeated a large number of times $\boldsymbol{n}$ and the number of successes of some event $\boldsymbol{A}$ were $\boldsymbol{N_A}$, the probability of occurrence for the event $\boldsymbol{A}$ is $\boldsymbol{p(A)}$:

$$p(A) = \lim_{n\to\infty}\left(\frac{N_A}{n}\right) \text{ [3]}$$

The following lines of code assign probabilities for the source generated characters using the relative frequency definition of probability.

```
for j=1:7
a(1,j)=length(find(DoupleInputText(j)==DoupleInputText));
ProbabilitiesOfCharacters(1,j)=a(1,j)/7;
end
```

These lines of code count the number of occurrences for each input letter in the whole input message and use the relative frequency definition of probability to assign probabilities for each letter.

- Although this definition is correct, it doesn't give probabilities that we usually encounter in practice for such a short input (only seven letters). Therefore, random probabilities are assigned to each input letter using the following lines of code.

```
ProbabilitiesOfCharacters = zeros(1,7);
```

```
for i = 1 : 7
ProbabilitiesOfCharacters(i) = rand(1)+ProbabilitiesOfCharacters(i); end
ProbabilitiesOfCharacters =
ProbabilitiesOfCharacters./sum(ProbabilitiesOfCharacters);
```

These probabilities are used throughout the report.

2.3 The n-gram Probabilities Matrix Generator

To maximize the objective presented in (3), **n** symbols are grouped and encoded simultaneously. This process decreases the average codeword length but doesn't affect the entropy since it is a property of the original message. Hence, the efficiency increases. In order to generate the nth gram probabilities vector, the following recursive formula is used. See appendix C for proof.

$$p_n = [C_1 * p_{n-1}, C_2 * p_{n-1}, C_3 * p_{n-1}, C_4 * p_{n-1}, C_5 * p_{n-1}, C_6 * p_{n-1}, C_7 * p_{n-1}] \dots (5).$$

The following lines of code do so:

```
POC= ProbabilitiesOfCharacters;
POC= [POC zeros(1,16800)];
lop=length(POC);
for a=1:4
lop1=length(POC(POC(a,:)>0));
NumberOfZeros=lop-7^(a+1);
POC(a+1,:)=[POC(1,1).*POC(a,1:lop1),POC(1,2).*POC(a,1:lop1),POC(1,3).*POC(a,1:lop1),
...
    POC(1,4).*POC(a,1:lop1),POC(1,5).*POC(a,1:lop1),POC(1,6).*POC(a,1:lop1),...
    POC(1,7).*POC(a,1:lop1),zeros(1,NumberOfZeros)];
end
ProbabilitiesOfCharacters=POC;
```

- The first and last lines of code are implemented only for presentation and convenience purposes.

2.4 Algorithm Design

The main objective of the designed algorithm is to generate codeword for each encoded character of the source's characters using Huffman coding technique. The flow chart shown below summarizes what the algorithm does. Each subprocess will be discussed in a dedicated section extensively.

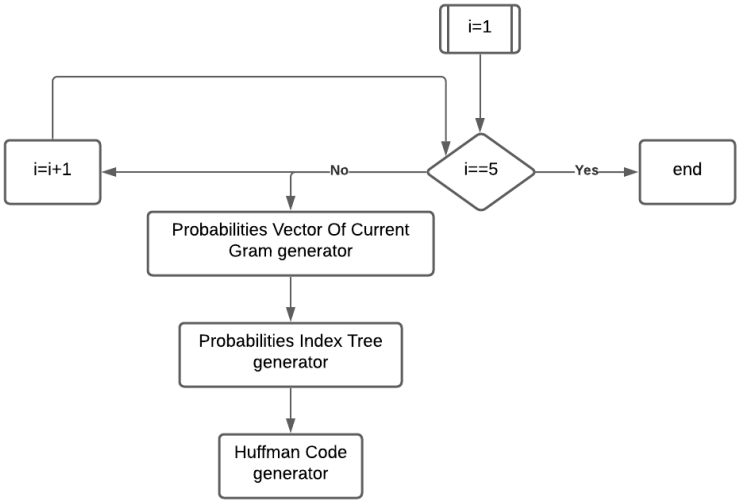


Fig.2. *Flow chart of the designed algorithm.*

2.4.1 Probabilities Index Tree generator

The idea behind the probabilities index tree generator is to generate the index matrix containing the corresponding indices of each element of the probability matrix. The following flow chart along with the following lines of code show the algorithm in which the probabilities index tree is generated. The second to fourth three lines of code are responsible for fetching the corresponding probability row for each n-gram, with only the non-zero elements and calculating the average of its length. This section includes three for-loops, the first one is dedicated to generating the indices of the probabilities matrix, which is arranged in ascending order in each row, starting with the index with the lowest probability as the first element in the row and ending with the index with the highest probability as the last element in the row, so that the first row of this matrix forms the final branches for the probabilities index tree. This tree is obtained by adding the two lowest probabilities in each row, which are the probability in the first (lowest one) cell and the probability in the second (highest one) cell of each row. When going backwards from the branches to the roots, the sum of these two elements is entered in the first cell in the next row, the remaining cells of the row are filled with the remaining probabilities from the previous row and then the arrangement process is repeated with each loop cycle until the final matrix is obtained with the end of the 1st sub-loop. (Lines 23-34)

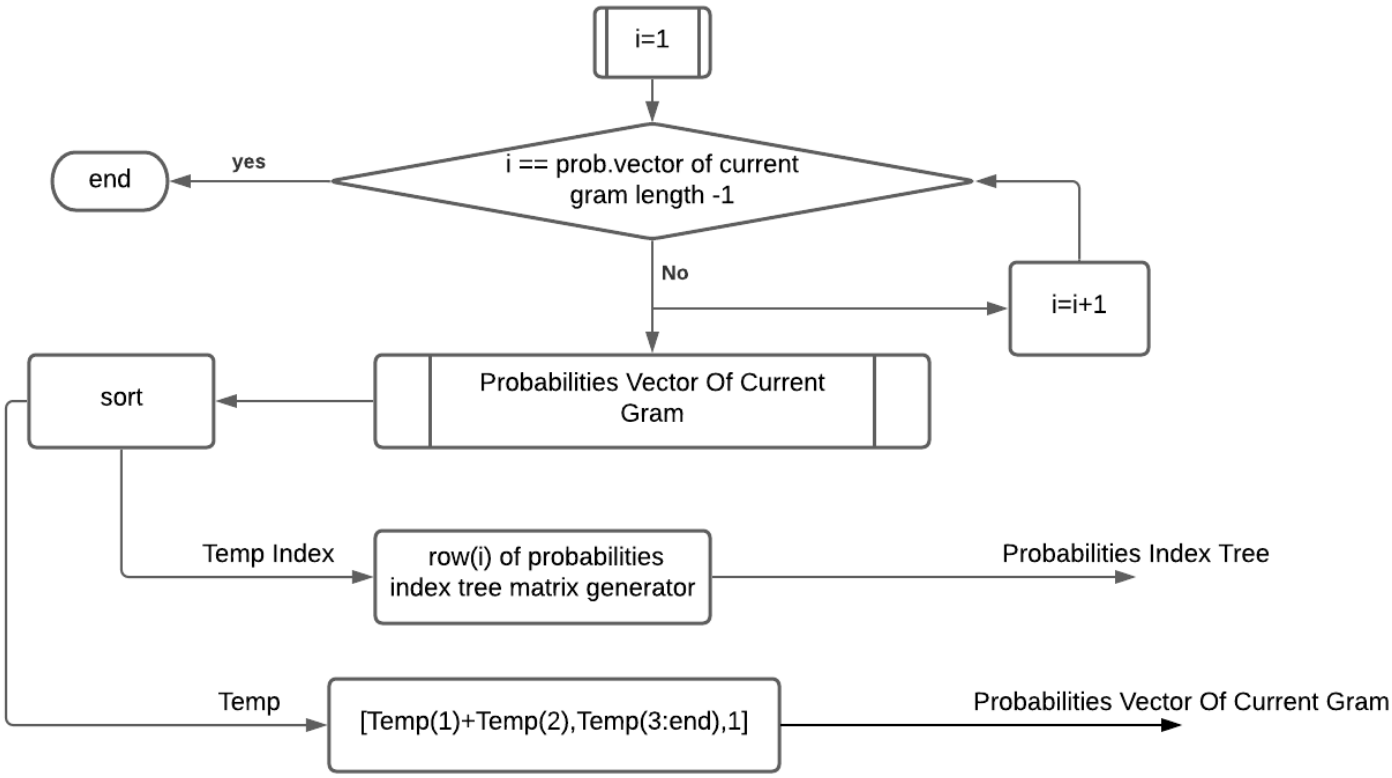


Fig.3. *Flow chart of probabilities index tree generator.*

```
for GramNumber=1:4
pvn=ProbabilitiesOfCharacters(GramNumber,:);
ProbabilitiesVectorOfCurrentGram=pvn(pvn>0);
l=length(ProbabilitiesVectorOfCurrentGram);
Temp=ProbabilitiesVectorOfCurrentGram;
ProbabilitiesIndexTree=zeros(l,l);
for i=1:l-1
[Temp,l0]=sort(Temp);
ProbabilitiesIndexTree(i,:)=[l0(1:l-i+1),zeros(1,i-1)];
Temp=[Temp(1)+Temp(2),Temp(3:end),1];
End
```

2.4.2. The Huffman Code Generator

After the index matrix is formed, the symbols 0 and 1 are distributed among the elements of the matrix (the index matrix may be referred to using the probabilities matrix because the position of the elements of each of both is the same and to communicate the idea more clearly), starting from the roots of the probability tree and ending with the branches. The higher probability in each row gives the symbol 1 and the immediately following probability the symbol 0.These two probabilities form

the roots (the lowest row in the probabilities matrix) of the tree. Then you go to the next lowest row, and the probability that similar to one of the probabilities in the previous row it gets the same symbol that the previous symbol had. The probabilities of the first two cells of each row are always summed (when walking from the branches to the roots), first: each of both was get the same probability symbol as the resulting probability from their addition was had, second: the symbol 0 is added to the first probability (in the first cell) and the symbol 1 is added to the second probability (in The second cell of the same row), and so on until the last branch in the tree is reached so that our temporary code tree is formed (lines 33 to 51).

```
TempCode=blanks(l*l)
TreeCode=blanks(l*l)
TempCode(l)='0'
TempCode(2*l)='1'
position0=0
position1=0
position2=0
for i=1:l-2
TreeCode=TempCode
position0=find(ProbabilitiesIndexTree(l-i,:)==1)*l
TempCode(1:l)=[TreeCode(position0-l+2:position0) '0']
TempCode(l+1:2*l)=[TempCode(1:l-1) '1']
for j=2:i+1
position2=find(ProbabilitiesIndexTree(l-i,:)==j)
TempCode(j*l+1:(j+1)*l)=TreeCode(l*(position2-1)+1:l*position2)
end
end
```

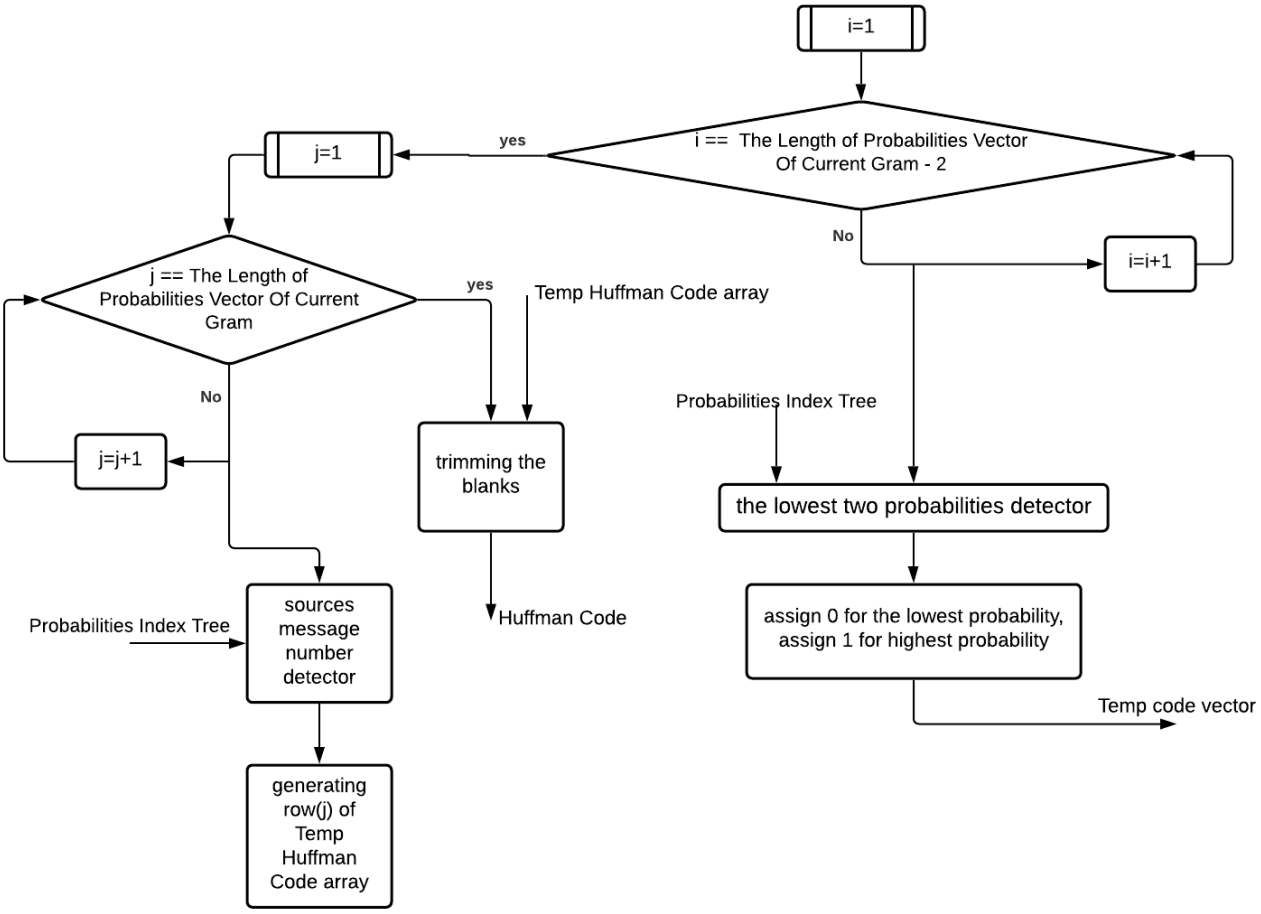


Fig.4. *Flow chart of Huffman Code generator.*

The last for Loop, it is intended to bring the final code from the temporary code tree, which represents the Huffman code for the message's characters, it arranged vertically so that each code corresponds to the letter it represents (lines 52 to 57).

```
TempHuffmanCode=char(zeros(l,l))
for i=1:l
position1=find(ProbabilitiesIndexTree(1,:)==i)
TempHuffmanCode(i,1:l)=TempCode(l*(position1-1)+1:position1*l)
end
HuffmanCode=strtrim(TempHuffmanCode)
```

2.5 Efficiency Calculator

To calculate the coding efficiency, we first need to calculate the average codeword length L and the entropy of the source H(S) using equations (1) and (2) respectively. In order to calculate the average codeword length, the length of each of the source's characters codeword L(i) is calculated by counting the number of binary bits in the encoded character, then equation (2) is applied. Once H(S) and L are known, the coding efficiency $\boldsymbol{\eta}$ is calculated using equation (3). This procedure is implemented using these lines of code:

```
for i=1:l
CodeLength(i)=length(find(abs(HuffmanCode(i,:))==48))+length(find(abs(HuffmanCode(i,
:))==49));
end
Entropy=sum(log2(ProbabilitiesVectorOfFirstGram).*ProbabilitiesVectorOfFirstGram);
AverageCodeLength=sum(CodeLength.*ProbabilitiesVectorOfCurrentGram)/GramNumber;
Effeciency=(Entropy/AverageCodeLength)*100;
```

The for loop counts the number of binary bits in each encoded character by searching for the ASCII values of the zeros and ones in the encoded character and measuring the length of the resultant vector. The remaining lines of code apply equations (1) to (3) respectively.

This procedure is done for each gram's probabilities vector.

III. Performance Evaluation

In order to evaluate the performance of the system's model, the designed algorithm and the implemented Matlab code, we need to consider the coding efficiency for each encoded source extension. The following simulations are done only on the grams (1,2,3,4) because of memory space and run time limitations. I will present in here the input characters, the assigned probabilities, the sum of the assigned probabilities, the source generated codes, average codeword length, entropy and efficiency for the first gram only because the output of the higher extensions take too much space to be shown in one page. For full output please run the full code in appendix D in Matlab software.

```
InputText = '-H98v11'


 ProbabilitiesOfCharacters = 1×7
       0.1858    0.2717    0.2127    0.0402    0.1270    0.0653    0.0974

SummationOfP = 1


Number of gram: 1

     message codeword Probability

Output = '-    111   0.18578'
Output = 'H     10   0.27169'
Output = '9     01   0.2127'
Output = '8   0010   0.040213'
Output = 'v    110   0.127'
Output = '1   0011   0.065279'
Output = '1    000   0.097351'

Entropy = 2.5856

Average codeword length = 2.6211

Effeciency = 98.6447 %
```

From this output, first, we can see that the discrete memoryless source successfully generated seven random characters. Second, the probability assigner assigned random probabilities to the source generated characters that match what would be expected practically. Third, the source encoded characters along with their codewords and probabilities match what we expected theoretically. Finally, the calculated values of entropy, average codeword length and efficiency are correct.

To show the effect of the higher order extensions on the coding efficiency, a scatter plot of coding efficiency vs gram number is simulated.

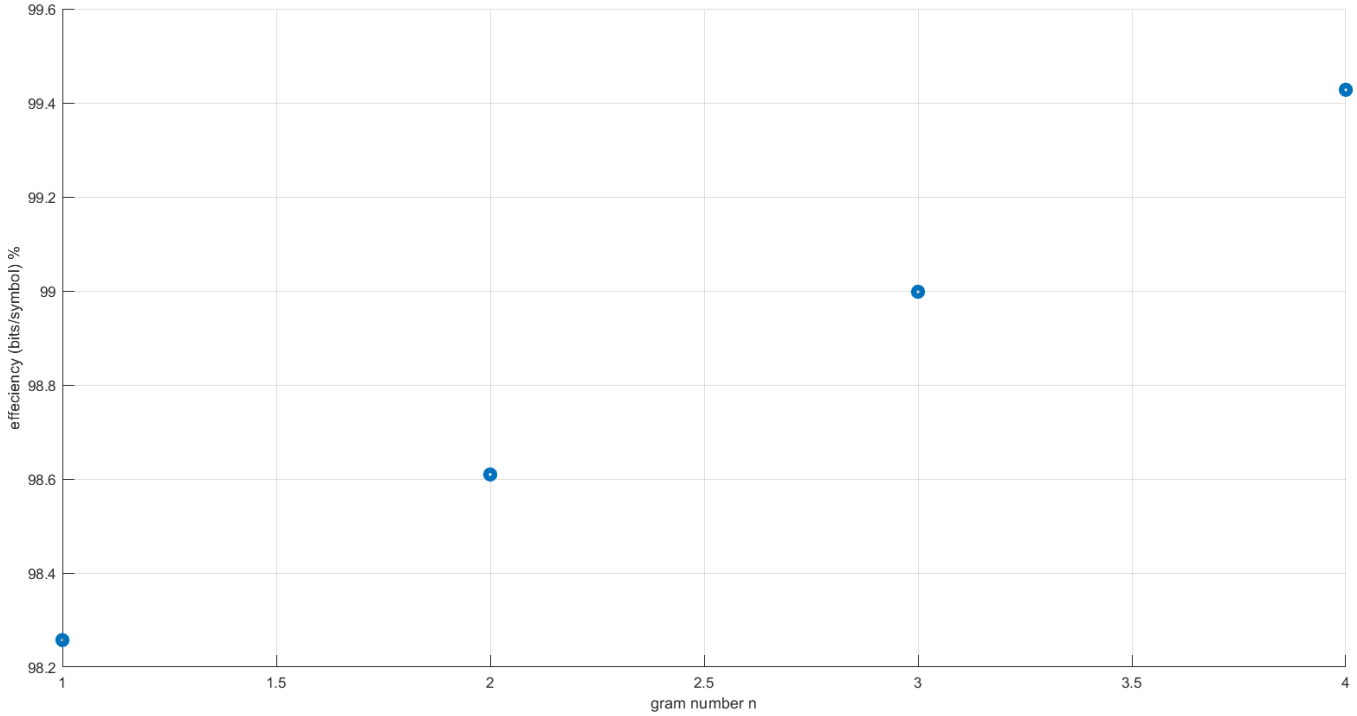
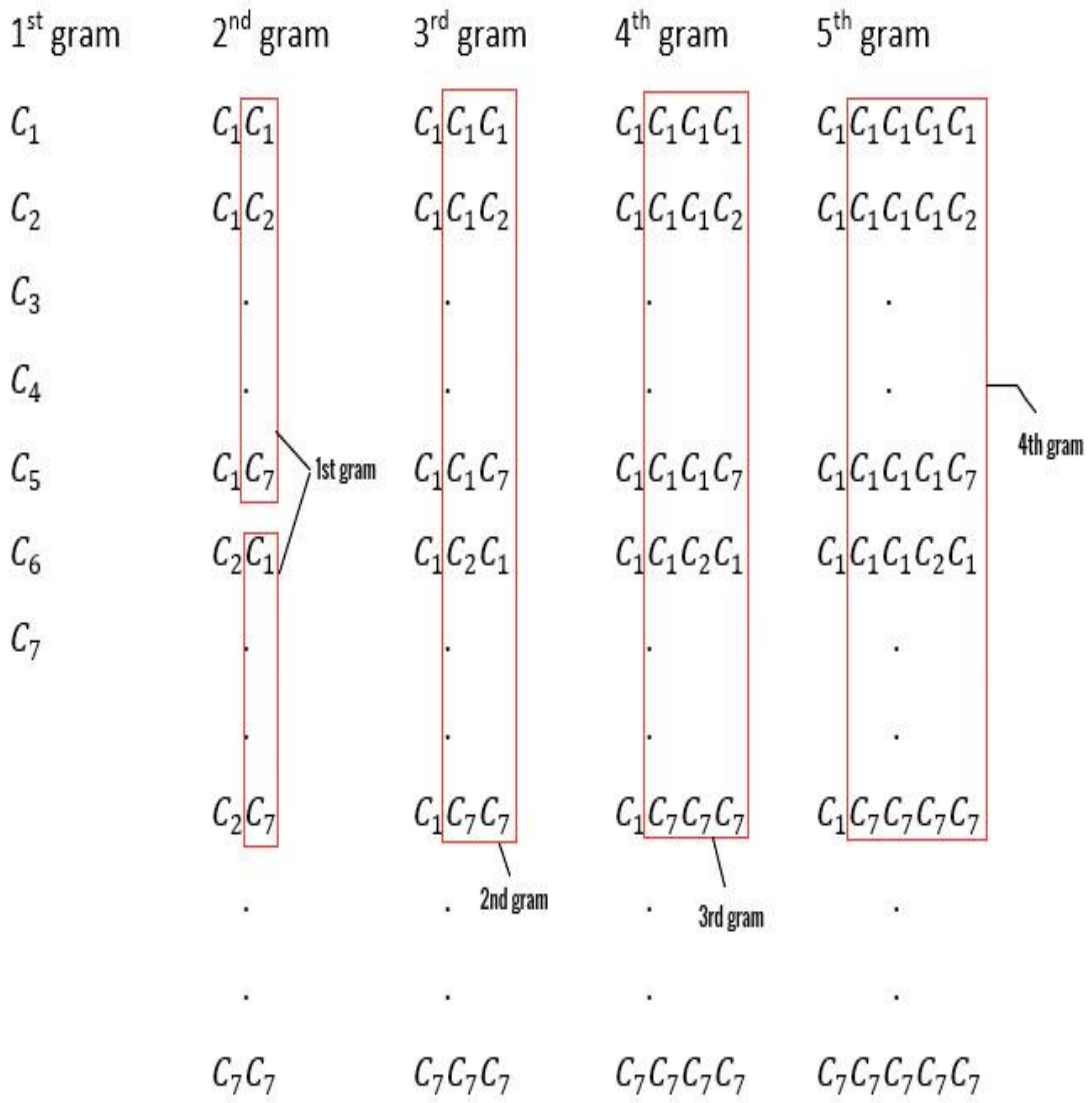


Fig.5. *Efficiency VS gram number plot*

IV. Conclusion and Discussion

In this project, we investigated the problem of lossless data compression with optimum efficiency using n-gram Huffman source coding technique. To do so, a low-complexity algorithm maximizing the efficiency has been proposed for Huffman codewords generation. The proposed algorithm was logically structured using flow charts and Matlab software. After simulating the system with random inputs, the algorithm proved its validity with figure (4) since we can clearly see that the coding efficiency increases as the gram number increase. There is one issue worth further exploration. The memory space needed for the algorithm is overly large for large gram numbers which takes a very long time to simulate. To overcome this issue, one might think of using the modified Huffman algorithm but what you gain in speed will be lost in efficiency. See Appendix E for more elaboration in the modified Huffman source coding technique.

APPENDIX A. PROVING THE N-GRAM PROBABILITIES MATRIX FORMULA



APPENDIX B. HUFFMAN MATLAP CODE

```
% The source
characters=(char(32:126));
DoupleInputText= randsample(32:126,7);
InputText=char(DoupleInputText);
```

```
% Probability assigner
POC=zeros(1,7);
a=zeros(1,7);
for j=1:7
a(1,j)=length(find(DoupleInputText(j)==DoupleInputText));
POC(1,j)=a(1,j)/7;
end
```

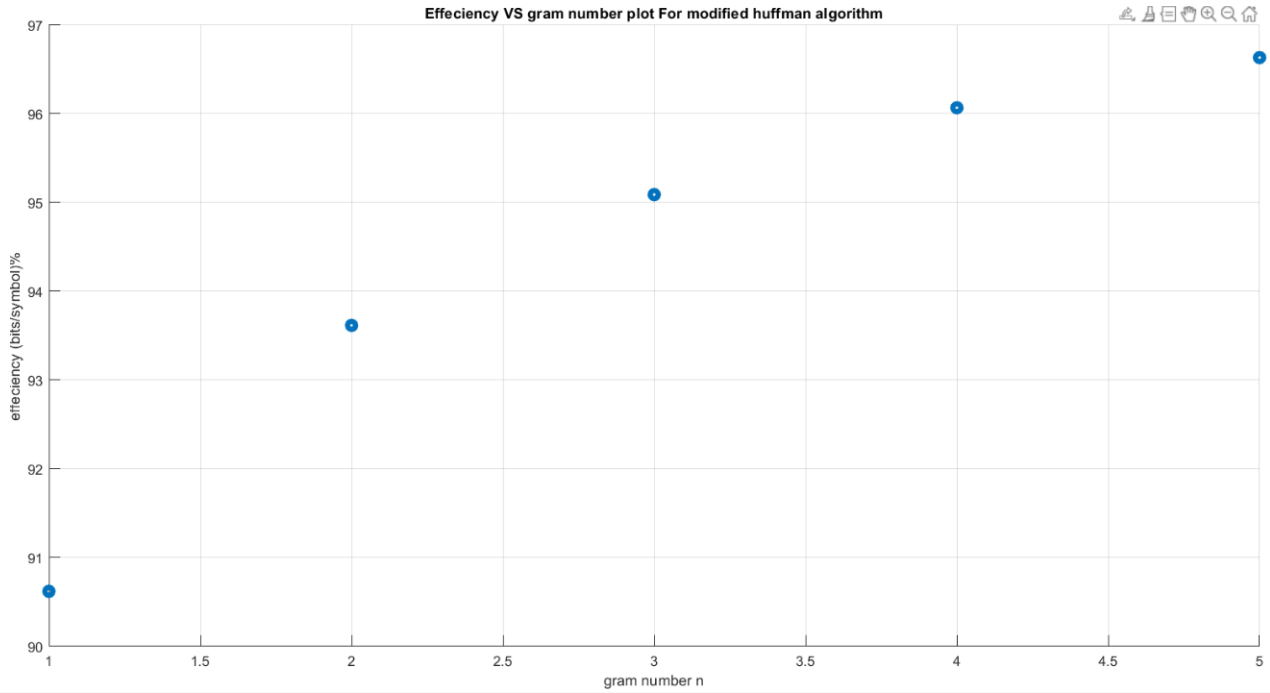
```
% n-gram propability matrix generator
POC=[POC zeros(1,16800)];
lop=length(POC);
for a=1:4
lop1=length(POC(POC(a,:)>0));
NumberOfZeros=lop-7^(a+1);
POC(a+1,:)=[POC(1,1).*POC(a,1:lop1),POC(1,2).*POC(a,1:lop1),POC(1,3).*POC(a,1:lop1),
...
    POC(1,4).*POC(a,1:lop1),POC(1,5).*POC(a,1:lop1),POC(1,6).*POC(a,1:lop1),...
    POC(1,7).*POC(a,1:lop1),zeros(1,NumberOfZeros)];
end
```

```
% The designed algorithm
for GramNumber=1:4
pvn=POC(GramNumber,:);
ProbabilitiesVectorOfCurrentGram=pvn(pvn>0);
l=length(ProbabilitiesVectorOfCurrentGram);
Temp=ProbabilitiesVectorOfCurrentGram;
ProbabilitiesIndexTree=zeros(l,l);
for i=1:l-1
[Temp,l0]=sort(Temp);
ProbabilitiesIndexTree(i,:)=[l0(1:l-i+1),zeros(1,i-1)];
Temp=[Temp(1)+Temp(2),Temp(3:end),1];
end
TempCode=blanks(l*l);
TreeCode=blanks(l*l);
TempCode(l)='0';
TempCode(2*l)='1';
position0=0;
position1=0;
position2=0;
for i=1:l-2
TreeCode=TempCode;
position0=find(ProbabilitiesIndexTree(l-i,:)==1)*l;
TempCode(1:l)=[TreeCode(position0-l+2:position0) '0'];
TempCode(l+1:2*l)=[TempCode(1:l-1) '1'];
```

```
for j=2:i+1
position2=find(ProbabilitiesIndexTree(l-i,:)==j);
TempCode(j*l+1:(j+1)*l)=TreeCode(l*(position2-1)+1:l*position2);
end
end
TempHuffmanCode=char(zeros(l,l));
for i=1:l
position1=find(ProbabilitiesIndexTree(1,:)==i);
TempHuffmanCode(i,1:l)=TempCode(l*(position1-1)+1:position1*l);
end
HuffmanCode=strtrim(TempHuffmanCode);
```

```
% Effeciency Calculator
disp('************************************************************')
disp(['Number of gram: ' num2str(GramNumber)])
disp(['     ','message',' --> ',' codeword',' Probability']);
CodeLength=zeros(1,l);
for i=1:l
CodeLength(i)=length(find(abs(HuffmanCode(i,:))==48))+length(find(abs(HuffmanCode(i,
:))==49));
Output=['X',num2str(i),'    ',HuffmanCode(i,:),'
',num2str(ProbabilitiesVectorOfCurrentGram(i))]
end
AverageCodeLength=sum(CodeLength.*ProbabilitiesVectorOfCurrentGram)/GramNumber;
PVOFG1=POC(1,:);
ProbabilitiesVectorOfFirstGram=PVOFG1(PVOFG1>0);
Entropy=sum(-
log(ProbabilitiesVectorOfFirstGram).*ProbabilitiesVectorOfFirstGram)./log(2);
Effeciency(1,GramNumber)=(Entropy/AverageCodeLength)*100;
display(['Entropy = ',  num2str(Entropy)])
display(['Average codeword length = ', num2str(AverageCodeLength)])
disp(['Effeciency = ', num2str(Effeciency(1,GramNumber)),' %'])
end
HorizontalAxis=[1 2 3 4];
scatter(HorizontalAxis,Effeciency,"LineWidth",4)
title('Effeciency VS gram number plot For huffman algorithm')
xlabel('gram number n')
ylabel('effeciency (bits/symbol)')
grid
```

APPENDIX C. SHANON-FANO Results



APPENDIX D. SHANON-FANO MATLAP CODE (TO SHOW THE 5-GRAM OPERATION)

```
%generating the random probability vector of the randomly generated
%characters
characters=(char(32:127));
r = randsample(32:127,7);
Text = strcat(r);
charactersLength = size(characters,2);
TextLength = size(Text,2);
poc = zeros(charactersLength,1);
for i = 1 : charactersLength
  for j=1 : TextLength
   if Text(j) == characters(i)
     poc(i) = rand(1)+poc(i);
   end
  end
end
poc = poc ./ sum(poc);
p=poc(poc>0)';
p=[p zeros(1,16800)];
%Generating the n-gram probability vectors
n11=length(p);
for a=1:4
m11=length(p(p(a,:)>0));
b=n11-7^(a+1);
p(a+1,:)=[p(1,1).*p(a,1:m11),p(1,2).*p(a,1:m11),p(1,3).*p(a,1:m11),p(1,4).*p(a,1:m11
),p(1,5).*p(a,1:m11),p(1,6).*p(a,1:m11),p(1,7).*p(a,1:m11),zeros(1,b)];
end
pmm=p(1,:);
pm=pmm(pmm>0);
```

```
entropy=sum(-log(pm).*pm)./log(2);
% n-gram calculations
for ng=1:4
p0=p(ng,:);
p1=p0(p0>0);
n=length(p1);
q=p1;
probabilityIndexTree=zeros(n,n);
for i=1:n-1
[q,l]=sort(q);
probabilityIndexTree(i,:)=[l(1:n-i+1),zeros(1,i-1)];
q=[q(1)+q(2),q(3:end),1];
end
p00=p(ng,:);
p11=p00(p00>0);
nn=length(p11);
qq=p11;
probabilitytree=zeros(n,n);
for i=1:n-1
[qq,ll]=sort(qq);
probabilitytree(i,:)=[qq(1:n-i+1),zeros(1,i-1)];
qq=[qq(1)+qq(2),qq(3:end),1];
end
% huffman code word generation
transientCode=blanks(n*n);
treeCode=blanks(n*n);
transientCode(n)='0';
transientCode(2*n)='1';
position0=0;
position1=0;
position2=0;
for i=1:n-2 % من هون
treeCode=transientCode;
position0=find(probabilityIndexTree(n-i,:)==1)*n;
transientCode(1:n)=[treeCode(position0-n+2:position0) '0'];
transientCode(n+1:2*n)=[transientCode(1:n-1) '1'];
for j=2:i+1
position2=find(probabilityIndexTree(n-i,:)==j);
transientCode(j*n+1:(j+1)*n)=treeCode(n*(position2-1)+1:n*position2);
end
end % لهون... بطيئة اللوغاريثمية فبدنا نشوف طريقة أسرع تجيب نفس النتيجة
HuffmanCode1=char(zeros(n,n));
for i=1:n
position1=find(probabilityIndexTree(1,:)==i);
HuffmanCode1(i,1:n)=transientCode(n*(position1-1)+1:position1*n);
end
% displaying the output
HuffmanCode =strtrim(HuffmanCode1);
disp('************************************************')
disp(['number of gram: ' num2str(ng)])
disp(['     ','message',' --> ',' codeword',' Probability']);
codeLength=zeros(1,n);
for i=1:n
```

```
codeLength(i)=length(find(abs(HuffmanCode(i,:))==48))+length(find(abs(HuffmanCode(i,
:))==49));
aa=['x',num2str(i),'    ',HuffmanCode(i,:),'    ',num2str(p1(i))]
end
avg_length=sum(codeLength.*p1)/ng;
effeciency(1,ng)=(entropy/avg_length)*100;
display(['Entropy = ',  num2str(entropy)])
display(['Average codeword length = ', num2str(avg_length)])
disp(['Effeciency = ', num2str(effeciency(1,ng)),' %'])
end
you=[1 2 3 4];
scatter(you,effeciency,"LineWidth",4)
title('Effeciency VS gram number plot For huffman algorithm')
xlabel('gram number n')
ylabel('effeciency (bits/symbol)')
grid
disp('Modified huffman calculations')
for ng1=1:5
v=p(ng1,:);
ss=v(v>0);
ss=sort(ss,'descend');
siling=log2(1/ss(1));
siling=round(siling,1,'significant');
sf=0;
fano=0;
if ng1==1
%initializations for Pk
n=1;Hx=0; %initializations for entropy H(X)
for i=1:length(ss)
Hx=Hx+ ss(i)*log2(1/ss(i)); %solving for entropy
end
end
disp('**************************************************')
disp(['number of gram: ' num2str(ng)])
disp(['Modified Huffman codes']);
for k=1:length(ss)
info(k)=-(log2(ss(k))); %Information
end
for j=1:length(ss)-1
fano=fano+ss(j);
sf=[sf 0]+[zeros(1,j) fano]; %solving for Information for every codeword
siling=[siling 0]+[zeros(1,j) ceil(log2(1/ss(j+1)))]; %solving for length every
codeword
end
for r=1:length(sf)
esf=sf(r);
for pb=1:siling(r)
esf=mod(esf,1)*2;
h(pb)=esf-mod(esf,1); %converting Pk into a binary number
end
hh(r)=h(1)*10^(siling(r)-1); %initializtion for making the binary a whole number
for t=2:siling(r)
hh(r)=hh(r)+h(t)*10^(siling(r)-t);    %making the binary a whole number
end                                        %e.g. 0.1101 ==> 1101
```

```
end
c={'0','1'};
for i=1:length(hh)
u=1;                                        %converting the codes into a string
for t=siling(i):-1:1
f=floor(hh(i)/10^(t-1));                %1001 ==>1 (getting the first highest unit of
a number)
hh(i)=mod(hh(i),10^(t-1));              %1001 ==>001(eliminating the first highest
unit of a number)
if f==1
if u==1
d=c{2};                          %conversion part (num(1001) to str(1001))
else
d=[d c{2}];
end
else
if u==1
d=c{1};
else
d=[d c{1}];
end
end
codex{i,:}={d};
u=u+1;
end
disp([d]);
end
tao=siling(1)*ss(1); %initialization for codeword length
for u=1:length(ss)-1 %computing for codeword length
tao=tao+siling(u+1)*ss(u+1);
end
T=tao/ng1; %computing for average codeword length
B=[flipud(rot90(ss)),flipud(rot90(siling)),flipud(rot90(info))];
disp(['Probability','   Length','   Information'])
disp(B)
disp(['Entropy H(X) = ',num2str(Hx),'bits/symbol'])
disp(['Average length,L = ',num2str(T),'bits/symbol'])
efff(1,i)=((Hx/T)*100);
eff=efff(efff>0);
%Coding efficiency
disp(['Efficiency=',num2str(eff(1,ng1)),'%'])
end
y=[1 2 3 4 5];
scatter(y,eff,"LineWidth",4)
title('Effeciency VS gram number plot For modified huffman algorithm')
xlabel('gram number n')
ylabel('effeciency (bits/symbol)')
grid
```

REFERENCES

[1] “Digital Communication,” J. Proakis, 5th edition, 2007, MGraw-Hill.

[2] Ding, Zhi_ Lathi, Bhagwandas Pannalal - Modern digital and analog communication systems-Oxford University Press, USA (2018_2019)

[3] Probability, random variables, and random signal principles / Peyton Z. Peebles, Jr-. 4th ed. p. cm.