EC31002 – Digital Communication Theory

Name: *Rekha Lokesh*

Roll No: *19EC10052*

Assignment 1 – Ziv and Lempel Compression

Problem Statement:

Using the ***Universal Compression method*** presented by ***Ziv and Lempel***, Encode the given text file (First chapter from 'The Master and Margarita'). There are different symbols used in the text and there is ***no probabilistic structure available***. Make your dictionary and then assign codewords for ***different variable length symbol strings***. Then using that ***dictionary***, you must ***encode*** the data.
Store the ***compressed data*** in a text file. Find the ***compression ratio***.

Next, start with the compressed text. ***Build the dictionary*** again from the ***compressed text***. Then decode the entire text. ***Only information the decoder*** can know about ***the encoder is the that of the dictionary***

Brief Theory:

- The Lempel-Ziv data compression algorithms differ from the normal source coding algorithms. They use ***variable-to-variable-length codes*** in which both the ***number of source symbols encoded*** and the ***number of encoded bits per codeword*** are ***variable***. Moreover, the codes are time varying.
- They do not require prior knowledge of the source statistics, yet over time they adapt so that the average codeword length L per source symbol is minimized and move towards differential entropy H(X). Such algorithms are called universal algorithms.

LZ77 Algorithm:

The LZ77 algorithm compresses a sequence x = x1, x2, ... from some given discrete alphabet χ of size M = $|\chi|$. At this point, no probabilistic model is assumed for the source, so x is simply a sequence of symbols, not a sequence of random symbols. A subsequence ($x_m, x_{m+1}, \ldots, x_n$) of x is represented by $(x)_m^n$. The algorithm keeps the ***w most recently encoded source symbols in memory***. This is called a ***sliding window of size w***. The number w is large and can be thought of as being in the range of 2^{10} to 2^{20}. The parameter w is chosen to be a power of 2. ***Both complexity and, typically, performance increase with w.***

LZ77 Algorithm:

1. ***Encode*** the ***first w symbols*** in a ***fixed-length code*** without compression.
2. ***Set the pointer P = w***. (This indicates that all symbols up to and including x_P have been encoded.)
3. ***Find*** the ***largest n***>=2 such that $x_{P+1}^{P+n} = x_{P+1-u}^{P+n-u}$ for some u in the range 1<=u<=w. The string x_{P+1}^{P+n} is encoded using n and u. Note that the string and its match can overlap. ***If no match exists for n>=2***, then, independently of whether a match exists for n= 1, set n= 1 and ***directly encode the single source symbol x_{P+1} without compression***.
4. ***Encode*** the integer ***n*** into a codeword from the ***unary-binary code***. In the unary-binary code, a positive integer nis encoded into the binary representation of n, preceded by a prefix of ceil ($\log_2$ n) zeroes.
5. ***If n>1, encode*** the positive integer ***n<=w*** using a ***fixed-length code of length log w bits***. (At this point the decoder knows n and can simply count back by u in the previously decoded string to find the appropriate n-tuple, even if there is overlap as above.)
6. ***Set*** the pointer ***P to P+n*** and go to ***step (3).*** (Iterate forever till the end of sequence).

It can be seen that the above encoding gives prefix free codes and the probability of a typical source string x^n for a Markov source is approximately $2^{-nH[X|S]}$. If w>> $2^{nH[X|S]}$, then, according to the previous item, $N_x^n \approx wp_X^n(x^n)$ should be large and x^n should occur in the window with high probability. Alternatively, if w<< $2^{nH[X|S]}$, then x^n will probably not occur. Consequently, the match will usually occur for n≈ (log w)/H[X|S] as w becomes very large.

Performance analysis:

Approach:

- At first, all the ***symbols*** are ***read from the text*** file using ***'readlines'*** command and a character array ***'src'*** consisting of the symbols (or alphabets) is formed as the source file (symbols in the txt file).
- There are many ways of constructing a dictionary. In my case, I focused mainly on constructing a ***dynamic (adaptive) dictionary*** whose size according to the given input.
- A dynamic dictionary ***'dict'*** is formed using mapping of keys and values where the ***keys*** are ***unique and distinct symbols*** of the given ***source file*** and values are ***fixed length binary codes*** where each ***codeword*** is of ***length log2(w)*** where w is the window length.

- Let's choose the window size w = 2^{13}. So, all the ***first w symbols*** are encoded at first using ***fixed length encoding*** where in the ***for every symbol***, ***1 is encoded first followed by the corresponding value of the symbol in dictionary***.
- In every iteration, sliding the window of length w, n and u are found out according to LZ77 algo and n and u are encoded using unary-binary encoding and fixed length encoding respectively. n followed by u is the order of encoding.
- In case when n = 1, the corresponding symbol is encoded using fixed length encoding as above.
- The encoded sequence ***'encd'*** is ***stored*** in a ***text file***. As the working environment is simulator, the decoding is written as a function whose ***input arguments*** are the dictionary ***'dict'*** and the encoded sequence ***'encd'.***
- After encoding successfully and calculating the compression ratio, the program calls the decoding function ***"lnz_decode()"*** which decodes the passed encoded sequence according to the LZZ7 algo and returns the character array of decoded symbols ***'char_dec'***.
- This ***'char_dec'*** is compared with the initial source character array ***'src'*** to verify the decoding which will also be displayed in the command window. Next, the ***decoded character array of symbols*** is ***stored*** as a ***text file*** (which will be exact replica of the given source text file if the program runs correctly) drawing parallel lines to the real-world scenario.

Results:

- For the given text source file from MnM chapter 1, it is found that the ***total*** number of ***symbols*** are ***25109*** and there are 61 unique symbols occurring in the ***'src'*** character array. So, for encoding this text, the dictionary ***'dict'*** formed will be of size 61 with fixed length binary codes from 0 to 60 with length of each codeword being $\log_2(w)$.
- With ***Ziv Lempel Encoding choosing w = 2^{13}***, it is found out that the encoded sequence length to be 181040. Uncompressed data size would be when each sequence is of fixed length encoding which is 25109 × 13 = 326417.

  Compression Ratio = $\frac{Uncompressed\ Data\ Size}{Compressed\ Data\ Size}$ = $\frac{326417}{181040}$ = ***1.8030***

  which can also be interpreted as data got compressed by ***44.5372%*** ($\frac{1.803-1}{1.803} \times 100$ = 44.5372 %)

- The following table gives the details of the scheme for different values of w. The link for source file is here.

$Log_2(w)$	Uncompressed Size	Compressed size	Compression ratio	Compression Percentage	Encoded sequence link	Decoded Text link
9	*225981*	*130878*	*1.7267*	*42.0845*	enc_512	dec_512
10	*251090*	*126930*	*1.9782*	*49.4484*	enc_1024	dec_1024
11	*276199*	*128398*	*2.1511*	*53.5125*	enc_2048	dec_2048
12	*301308*	*141427*	*2.1305*	*53.0623*	enc_4096	dec_4096
13	*326417*	*181040*	*1.8030*	*44.5372*	enc_8192	dec_8192
14	*351526*	*277530*	*1.2666*	*21.0499*	enc_16384	dec_16384

```
Command Window
  Rem: 9
  Rem: 8
  Rem: 7
  Rem: 4
  Rem: 3
  Rem: 1
  Rem: 0
  w = 512, Uncompressed size = 225981, Compressed size = 130878, Compression Ratio is: 1.7267, Compressed % = 42.0845
  Decoding completed successfully and correctly
fx >>
```

```
Command Window
  Rem: 9
  Rem: 8
  Rem: 7
  Rem: 4
  Rem: 3
  Rem: 1
  Rem: 0
  w = 1024, Uncompressed size = 251090, Compressed size = 126930, Compression Ratio is: 1.9782, Compressed % = 49.4484
  Decoding completed successfully and correctly
fx >>
```

```
Command Window
  Rem: 13
  Rem: 11
  Rem: 8
  Rem: 7
  Rem: 4
  Rem: 1
  Rem: 0
  w = 2048, Uncompressed size = 276199, Compressed size = 128398, Compression Ratio is: 2.1511, Compressed % = 53.5125
  Decoding completed successfully and correctly
fx >>
```

***Fig1:* Simulated Result for $log_2(w)$ = {9, 10, 11} respectively**

```
Command Window
Rem: 13
Rem: 11
Rem: 8
Rem: 7
Rem: 4
Rem: 1
Rem: 0
w = 4096, Uncompressed size = 301308, Compressed size = 141427, Compression Ratio is: 2.1305, Compressed % = 53.0623
Decoding completed successfully and correctly
fx >>
```

```
Command Window
Rem: 18
Rem: 11
Rem: 8
Rem: 7
Rem: 4
Rem: 1
Rem: 0
w = 8192, Uncompressed size = 326417, Compressed size = 181040, Compression Ratio is: 1.8030, Compressed % = 44.5372
Decoding completed successfully and correctly
fx >>
```

```
Command Window
Rem: 21
Rem: 11
Rem: 8
Rem: 7
Rem: 3
Rem: 1
Rem: 0
w = 16384, Uncompressed size = 351526, Compressed size = 277530, Compression Ratio is: 1.2666, Compressed % = 21.0499
Decoding completed successfully and correctly
fx >>
```

***Fig2:** Simulated Result for $\log_2(w) = \{12, 13, 14\}$ respectively*

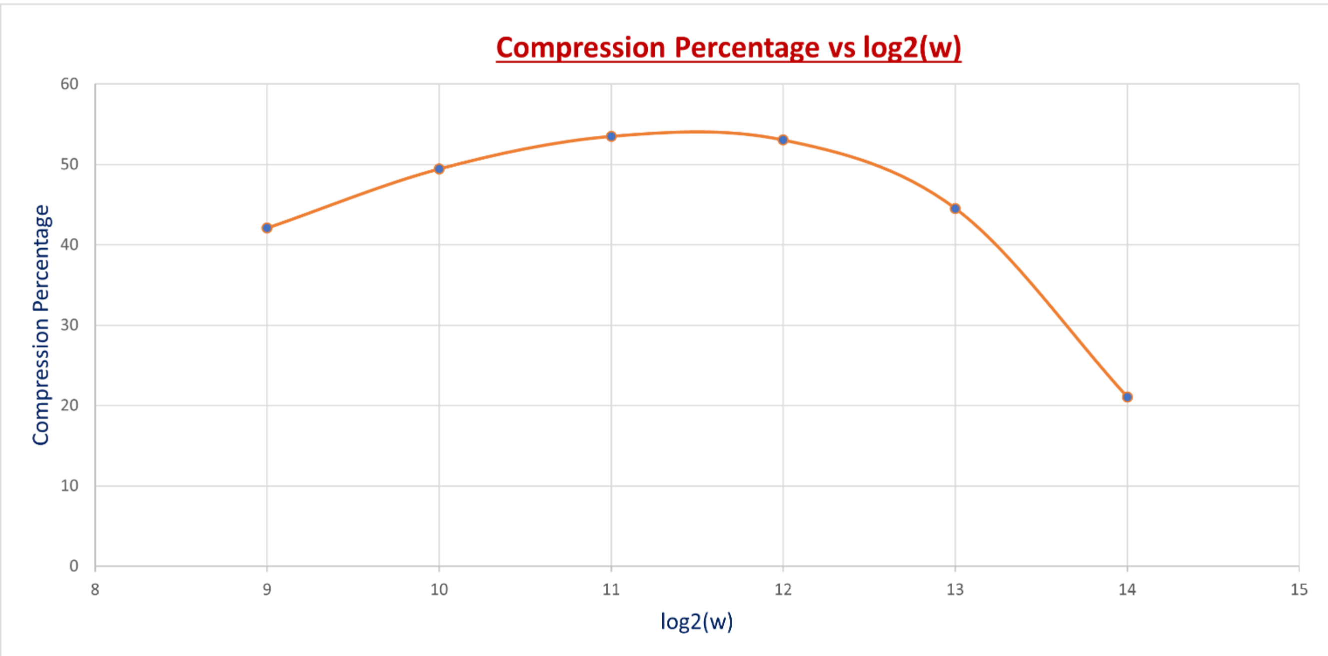

***Fig3:** Compressed Ratio vs log2(w)*

Discussion:

- In the Lempel and Ziv Algo LZ77, ***Encoding*** takes time, which is ***inversely proportional*** to the ***window length***, but decoding is done quickly. Encoding takes much time because of finding largest n and corresponding u in the LZ77 algo which is actually the key point of the algorithm.

  Note: For the given source file MnM chapter 1, my program takes around 5 mins to give the result for w = 2^{14} and around 8 mins for w = 2^{13} and so on. The progress can be tracked from the console as Rem goes to zero when the program is about to finish.
- ***Correct decoding*** proves the ***prefix free nature of the Ziv and Lempel encoding***
- Generally, as w increases, ***complexity of analysis increases, and performance also increases rapidly with increase in w*** since as w increases sliding window length increases and so for each iteration finding largest n becomes easy but there will be more computation in one iteration.
- So, a ***trade off*** is to be brought ***balancing*** the ***complexity of analysis*** and ***performance***. In my program, I ***dealt*** it keeping using the ***variable*** called ***compression ratio*** defined according to the given problem statement.
- This ***compression ratio*** had a peak at ***w = 2^{11}*** but w = 2^{12} or w = 2^{13} seem better as it is relatively faster (better performance) with just a slight drop in the compression percentage.
- The ***main advantages*** of the Lempel and Ziv approach of source encoding against other encoding methods are:
  - ***No prior knowledge*** required about the ***probability of occurrence of each symbol*** as in case of Huffman encoding scheme.
  - ***Compression ratio*** at the encoding side can be ***varied*** according to a given application just by changing the ***window length*** without altering the source file as the ***dictionary adapts*** to the change (length of the values in dictionary change) and ***no extra information*** is necessary to be provided at the ***decoding site.***
  - Thus, $\bar{L}$ (Expected Codeword length) can be ***brought close to H(X)*** (by choosing appropriate window length) ***resulting in efficient source coding***.
- Thus, LZ77 algo is carried out and verified according to the problem statement in MATLAB environment.

For Further Reference:

MATLAB FILE LINK – *MATLAB File Link for Assignment 1*

FULL – *Complete Folder for Assignment 1*

MATLAB CODE:

```
%   REKHA LOKESH
%   19EC10052
%   Digicomm Assignment 1

clear all;
close all;
clc;

src_path = "MnM_source_file.txt";    %   Path for the Source Text File
src_temp = readlines(src_path); %    Importing the text as string array

src_str = "";
for i=1:size(src_temp, 1)
    src_str = src_str + src_temp(i) + newline;    %   Writing the whole string
array as single string where is for newline
end

src = char(src_str);   %Converting string to char vec

w = 2^13; %Length of the sliding window

%   --------    Preparing Dictionary   ---------
uniq = unique(src); %Finding the unique occurences of characters in the string

%Let the dictionary consists of the unique ASCII value characters in the
%string

M = size(uniq,2);   %Number of unique characters
key_vec = repmat(cellstr(char('a')), M, 1); %Creating the keyvec to have only
the number of unique occurences
key_vec(1, :) = cellstr(string(uniq(1)));
bin_size = log2(w);
value_vec = repmat(convertCharsToStrings(dec2bin(0, bin_size)), M, 1);
for i = 1:(M-1)
   key_vec(i+1, :) = cellstr(string(uniq(i+1)));
   value_vec(i+1, :) = (convertCharsToStrings(dec2bin(i, bin_size)));
end

dict = containers.Map(transpose(key_vec), transpose(value_vec));   %Preparing
the dictionary is done


%   --------    ENCODING STARTS   ---------
%Encoding the first w symbols of src using fixed length encoding
encd = "";   %String that stores the encoded values
for i = 1:min([w, size(src, 2)], [], 'all')
    encd = encd + "1" + string(dict(string(src(i))));   %Fixed length encoding
is done with n = 1
end

P = w;  %Pointer is set to w

disp("Size of src is: "+int2str(size(src, 2)));
src_size = size(src, 2);
while (P<src_size)
    [n, u] = findlargestmatch(src, P, w);
    if (n==1)
        encd = encd + "1" + string(dict(string(src(P+1))));   %Fixed length
encoding is done
    else
        temp1 = dec2bin(n, 2*floor(log2(n))+1); %n codeword Unary Binary
Encoding
```

```
        temp2 = dec2bin(u, log2(w)); %u codeword Fixed length Encoding
        encd = encd + string(temp1) + string(temp2);    %Encoded n, u
    end
    P = P + n;
    disp("Rem: "+int2str(src_size-P));    %To see the progress of the simulation
end

%Writing encoded data in a txt file
fileID = fopen("Encoded_Data_w="+int2str(w)+".txt", 'w');
fprintf(fileID, encd);
fclose(fileID);
comp_ratio = size(src, 2)*bin_size/strlength(encd);  %since Compression Ratio = Uncompressed Size/Compressed Size
comp_percent = (comp_ratio-1)/comp_ratio*100;
disp("w = " + int2str(w) + ", Uncompressed size = " + int2str(src_size*bin_size) + ", Compressed size = " + strlength(encd) + ", Compression Ratio is: "+sprintf("%.4f", comp_ratio)+", Compressed % = "+sprintf("%.4f", comp_percent));
%Encoding Ends

%   --------    DECODING STARTS   ---------
% Calling the Decoding Function
char_dec = lnz_decode(encd, dict);   %Calling the decoding functions whose arguments are encoded string and dictionary which returns the decoded string
 if(char_dec==src)
    disp("Decoding completed successfully and correctly");
else
    disp("Decoding completed successfully and incorrectly");
end

%Printing the decoding symbols in text file
fileID = fopen("Decoded_text_file_w = "+int2str(w)+".txt", 'w');
m = 1;
sz = size(char_dec, 2);
while (m<=sz)
    q = 0;
    while ((m+q)<=sz && char_dec(m+q)~=char(newline))
       q = q + 1;   %Traversing the whole sentence until a newline character is detected
    end
    if(m+q+1 <= sz)
        fprintf(fileID, string(char_dec(m:(m+q-1)))+'\n');  %print newline if this is not the last string to be entered
    else
        fprintf(fileID, string(char_dec(m:(m+q-1))));
    end
    m = m+q+1;
end
fclose(fileID);



function [res1, res2] = findlargestmatch(src, P, w)
    sz = size(src, 2);
    if ((sz-P)<2)
        res1 = 1;
        res2 = 1;
    else
        res1 = 1;
        res2 = 1;
        for n = 2:(sz-P)
            substr = src(1, (P+1):(P+n));
            k = strfind((src(1, (P+2-w):(P+n-1))), substr);   %Since Range of u is 1 to w
```

```
            if(all(size(k)~= [0, 0], 'all') && (k(1, 1)+P-w)<=P)
               res1 = n;
               res2 = P-(k(1, 1)+P-w+1)+1;
            end
        end
    end
end

%   --------    DECODING Function   ---------
function dest_dec = lnz_decode(enc_src, dic)
    enc = char(enc_src); %Converting encoded string to char to access easily
    res = '';  %Initializing decoded char array
    val = values(dic);
    val_sz = strlength(string(val(1))); %finding the values size from the
dictionary
    dec_dict = containers.Map(values(dic), keys(dic));  %Inverse Mapping the
dictionary values
    i = 1;
    while (i<=size(enc, 2))
        if(all(enc(i)=='1', "all"))   %Fixed Length encoding is detected
            %Then detect the next w/val_sz bits from where the symbol can
            %be detected by reverse mapping
            i = i + 1;
            tmp = enc(i:(i+val_sz-1));
            symb = dec_dict(tmp);   %decoding the symbol as string
            res = char(string(res)+symb); %Concatenating the symbol to the
decoded string
            i = i + val_sz;
        else
           %It is a unary-binary code. So go on detecting zeroes until 1
           %appears. If k zeros detected then binary length of n is 2k+1
           k = 1; %number of zeros detected
           while (enc(i+k)=='0')
              k = k + 1;
           end
           n = bin2dec(enc(i:(i+2*k)));
           i = i + 2*k+1;
           u = bin2dec(enc(i:(i+val_sz-1)));
           p = size(res, 2);
           st = p-u+1;
           en = st+n-1;
           if(en<=p)    %no overlapping between string and symbols
                res = char(string(res)+string(res(st:en)));%Appending the string
           else          %overlapping between string and symbols
                symb = string(res(st:p));

                len = p-st+1;
                rem = n;
                stri = "";
                while (rem>=len)
                    stri = stri + symb;
                    rem = rem - len;
                end
                if(rem>0)
                   stri = stri + string(res(st:(st+rem-1))) ;
                end
                res = char(string(res)+stri);%Appending the string
           end
           i = i + val_sz;
        end
    end
    dest_dec = res;
end
```