**SPL-1 Project Report, 2018**

**reZ1**

**A simple compression algorithm tester**

**SE-306 : Software Project Lab-1**

Submitted by

***Md. Rezowan Talukder***

**BSSE Roll No: 0933**

**BSSE Session: 2016-2017**

Supervised by

***Md. Rayhanur Rahman***

**Lecturer**

**Institute of Information Technology**

**University of Dhaka**

****

**Institute of Information Technology**

**University of Dhaka**

Date: 27-05-2016

**Table of Contents**

1. Introduction ……………………………………………………………………………………… 3

1.1. Background Study……………………………………………………………………………… 4

1.1.1 Lampel Ziv Welch algorithm……………………………………………………………….… 4

1.1.2 Run Length Encoding………………………………………………..…………………………5

1.1.3 Huffman Encoding……………………………………………………………………………..7

1.1.4 Lempel Ziv 77 Encoding………………………………………………………………………8

1.2 Challenges……………………………………………………………………………………… 10

2. Project Overview…………………………..………………………………….…………………..10

3. User Manual Manual……………….…………………………………………………………….13

4. Conclusion……………………………………………………………………………………….14

References………………………………………………………………………………………..14

**1.Introduction**

The purpose of this software project is to determine an appropriate compression algorithm from various of compression algorithms by compression ratio. There are a lot of algorithms for Data compression. Every algorithm has some unique techniques thus different algorithm works well for different type of file. In this project my aim is to show the compression ratio of different algorithms for .txt file. This software will let user to choose a file from drive then this software will show the compression ratio using different algorithms of that file using Bar Diagram. This software is developed in Java; Java Swing is used for making software Interface and the API JFreeChart is used to make Bar Diagram.

Basicaly this this software is build focused on lossless compression algorithm. Four well-known algorithm’s compression ratio can be tested in this tool. Those algorithms are Lampel-Ziv-Welch(LZW) , Huffman Encoding , Run Length Encoding(RLE) and Lampel-Ziv-77(LZ77) .

A interface is build using Java Swing where user can choose file from drive. User also can extract compressed file using those four algorithms if he/she want. The name of the compressed file is extended by the algorithm name. For example the compressed file by LZW algorithm will be named like file.lzw , the compressed file by RLE algorithm will be named like file.rle e.t.c . But user can uncheck if he/she doesn’t want to extract the compressed files.

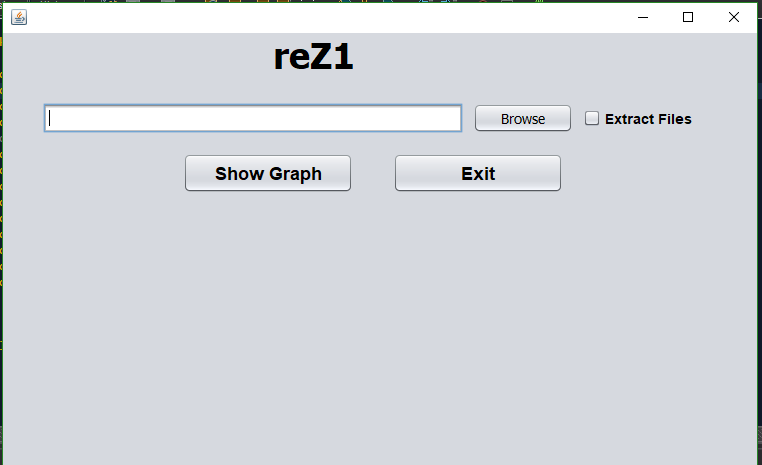


Figure: 1.1

* 1. **Background Study**
     1. **Lampel-Ziv-Welch (LZW) Compression**

In 1980, Terry Welch invented LZW algorithm which became the popular technique for general purpose compression systems. The major technique of LZW algorithm is to preload the dictionary with all possible symbols that can occur. LZW compression replaces string of characters with codes. LZW algorithm is a lossless data compression algorithm which is based on dictionaries [1]. This LZW compressor maintains records with characters that have been read from a file to be compressed. Each character is represented by an index number in the dictionary.

LZW compression algorithm is dictionary based algorithm which always output a code for a character. Each character has a code and index number in dictionary. Input data which we want to compress is read from file. Initially data is entered in buffer for searching in dictionary to generate its code. If there is no matching character found in dictionary then it will be entered as new character in dictionary and assign a code. If character is in dictionary then its code will be generated. Output codes have less number of bits than input data. This technique is useful for both graphics images and digitized voice.

**Here is the compression process of the LZW algorithm:**

*String j, char c;*

*j- get input character*

*while (there is still input character)*

*ch- transfer input string to ch .*

*if (ch is in dictionary) Generate its codeword;*

*else update ch and get next character to ch and*

*again search data in dictionary;*

*if ( it is not present in dictionary ) then add that string to dictionary;*

*end ;*

Now lets consider a string “BAABAABB” to compress using LZW algorithm. The output of this code is “---------” . In this example when input string (BAABAABBC) is given as a text to LZW compression algorithm. Initially every single character will save in buffer. When ‘B’ is move to buffer “parse string” then it will replace by a value of the next number. Then ‘BA’ is sent but it is also not yet in dictionary, so it will also get a value from dictionary. Like this if a prefix is matches with data existing in dictionary then it will get the value of the matches. Otherwise that prefix will given a new value.

**Here is the decompression process of the LZW algorithm:**

*ch = output code*

*while (there is still data to read)*

*code =get input character;*

*if (code is not in the dictionary) entry =get translation of code;*

*else entry=get translation of output code;*

*output entry;*

*ch =first character in entry*

*add output code + c to the dictionary*

*output code = code;*

*end ;*

* + 1. Run length Encoding(RLE)

RLE is a data compression technique widely used in TIFF, BMP and PCX file. An important characteristic of RLE is that it compresses any kind of information independent of its type, only difference will be the achieved compression ratio which depends on the content.

RLE uses a technique which looks for strings of repeated characters and thus bringing down the physical size by encoding these repeated strings. The encoding is applied on the repeated string (run), folding it in two bytes, amongst the two bytes the first one gives the number of characters called the run count containing 1-256 Characters. Taking as an example 16 '0' characters would require 8 bytes to store i.e. 0000000000000000. After compressing RLE requires 2 bytes: 08 00 here 08 is the count and 00 is the character string.

The block diagram for RLE algorithm is given as follow:

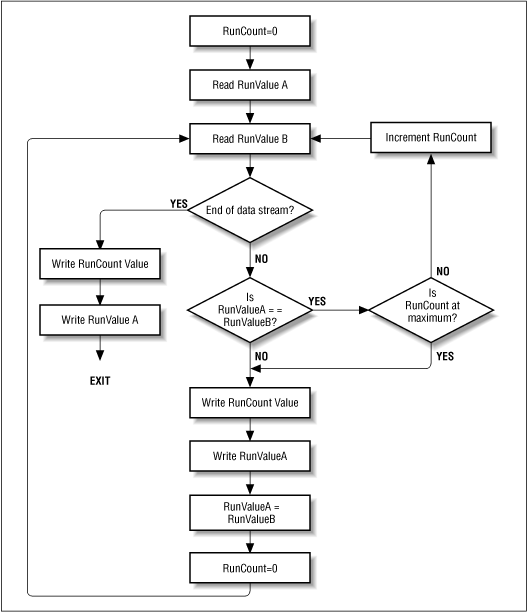


Figure: 1.2

But RLE works bad when the data stream doesn’t contain repeated bits. For example if a word is like “abcdefghij”, then compressing it by RLE algorithm we obtain “a1b1b1d1e1f1g1h1i1j1” which is twice than the actuall data. Thus whenever there are no repeating bit RLE should not used there. Image file like BMP,TIFF contain lots of repeated data, so compressing Image file RLE algorithm used vastly. Some more example are shown below:

|  |  |
| --- | --- |
| **Original Data** | **Compressed with RLE** |
| aabcccheeee | A2b1c3h1e4 |
| aaaaaaaaaabbbbbbbbbbbbbbb | A10b14 |
| erty | E1r1t1y1 |
| Aweaweaweaawweeaawwweee | A1w1e1a1e1a2w2e2a2w3e3 |

* + 1. **Huffman Codding**

Huffman encoding is a compression algorithm where define data elements with higher frequency are coded using special technique. Huffman encoding is based on a minimum coding technique. There are two types of prefix coding methods used for this purpose,

• Fixed length codes

• Variable length codes

As the name suggests fixed length coding refers to codes having the same length whereas the variable length code words are of varying length. Consider four elements a, b, c and d. The frequencies are 20, 10, 30 and 40. Hence fixed length and variable length codes can be given as follow-

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Coding** | **a** | **b** | **c** | **d** |
| **Frequency** | 20 | 10 | 30 | 40 |
| **Fixed length** | 000 | 001 | 010 | 011 |
| **Variable length** | 1000 | 101 | 10 | 1 |

Figure: 1.3

The fixed length coding takes 20x3 + 10x3 + 30x3 + 40x3 so the total is 300 bits whereas variable length coding takes 1x40 + 2x30 + 3x20 + 4x10 = 200 bits to represent the same data.

So,Variable length encoding is choosed a Huffman encoding. High number of bits can be reduced with Huffman coding as given in the following figure. The following figure is the implementation of the Huffman coding. The steps followed above are.

• Compute the complete frequency of all the data N.

• Start with the data element with the highest frequency going down to the lowest. As in the above figure starts with d and ends with a.

• Hence a compressed code will be obtained with variable-length codes for the data.

• compress code id obtained going from root to left and right and giving code 0 and 1 respectively.

1. 1

0 1

1. 1

Figure 1.x: Huffman encoding

* + 1. **Lampel Ziv 77 Encoding**

This algorithm is also another dictionary based lossless compression algorithm invented in 1977 by Jacob Ziv and Abraham Lampel. In the LZ77 approach, the dictionary work as a portion of the previously encoded sequence. The encoder examines the input sequence by pressing into service of sliding window which consists of two parts: Search buffer and Look-ahead buffer. A search buffer contains a portion of the recently encoded sequence and a look-ahead buffer contains the next portion of the sequence to be encoded.

The algorithm searches for the longest match with the beginning of the look-ahead buffer and outputs a pointer to that match. The sequence is encoded in the form of a triple <o, l, c>, where ‘o’ stands for an offset to the match, ‘l’ represents length of the match, and ‘c’ denotes the next symbol to be encoded.

**Here is the encoding technique for this algorithm:**

*While (look-Ahead Buffer is not empty)*

*{*

*Get a pointer (position, length) to longest match;*

*if (length > 0) {*

*Output (position, Longest match length, next symbol);*

*Shift the window by (length+1) positions along;*

*}*

*Else {*

*Output (0, 0, first symbol in the look-ahead buffer);*

*Shift the window by 1 character along;*

*}*

*}*

**Here is an example of encoding process of this algorithm:**

String to be compressed: abracadabrad

Output Tuple: (Offset, Length, Next Symbol)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Search Buffer** | | | | | | | **Look Ahead Buffer** | | | | | |  | **Tuple** |
| **7** | **6** | **5** | **4** | **3** | **2** | **1** | **a** | **b** | **r** | **a** | **c** | **ad…..** |  | **(0,0,a)** |
|  |  |  |  |  |  | **a** | **b** | **r** | **a** | **c** | **a** | **dab..** |  | **(0,0,b)** |
|  |  |  |  |  | **a** | **b** | **r** | **a** | **c** | **a** | **d** | **abr…** |  | **(0,0,r)** |
|  |  |  |  | **a** | **b** | **r** | **a** | **c** | **a** | **d** | **a** | **brad** |  | **(3,1,a)** |
|  |  |  | **a** | **b** | **r** | **a** | **c** | **a** | **d** | **a** | **b** | **rad** |  | **(0,0,c)** |
|  |  | **a** | **b** | **r** | **a** | **c** | **a** | **d** | **a** | **b** | **r** | **ad** |  | **(2,1,a)** |
|  | **a** | **b** | **r** | **a** | **c** | **a** | **d** | **a** | **b** | **r** | **a** | **d** |  | **(0,0,d)** |
| **a** | **b** | **r** | **a** | **c** | **a** | **d** | **a** | **b** | **r** | **a** | **d** |  |  | **(7,4,a)** |

So, after compression we will get 8 tuple from 12 character. So the compression ratio is 12/8=1.5

**1.2 Challenges**

* The main challenge to complete this project was to learn 4 broad compression algorithms named Lampel-Ziv-Welch, Run Length Encoding. Huffman Encoding and Lampel-Ziv-77 algorithms and implement code for those algorithm.
* Using Java Swing to make user interface.
* JFileChosser API is used to choose file from hard drive and finally put compressed file where the original file was.
* Give different name for every compressed file related to algorithm name was another challenge. For example if the file name is XXX.txt then the compressed file from this tool is name like XXX\_huff.txt, XXX\_rle.txt, XXX\_lzw.txt and XXX\_lz77.txt. thus user could easily recognize which file is compressed by which algorithm.
* Another main challenge was to use the API JFreeChart to make Bar Diagram to show the compression ratio. JFreeChart needs a lot of parameter and I had to learn.
* Finally merging all the class in main was another challenge.

1. **Project Overview**

There are total 4 classes in this project including main class. The main class is named MainAndOutForm.java. In this class there is code for user interface and and creating object of other classes. Here is the figure of user Interface:

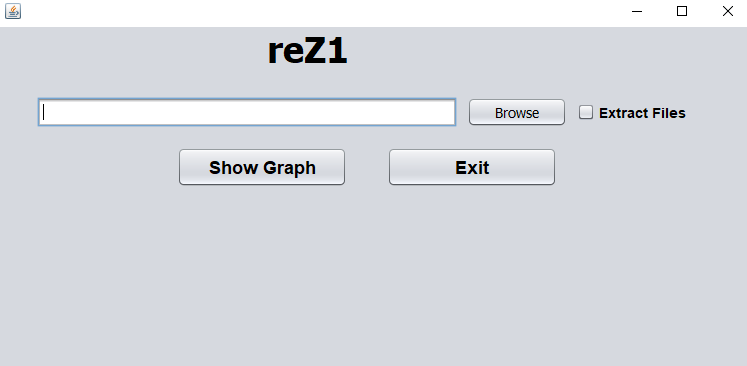


Figure 2.1

This portion of code let user to select a file from his drive and get the location of the to be compressed.

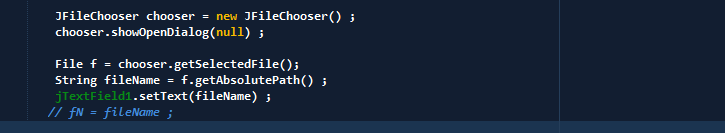


Figure: 2.2

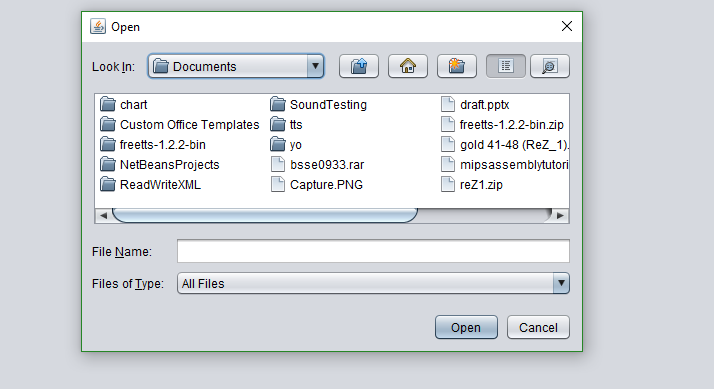


Figure:2.3

After selecting the file this tool creates instance of LZWCompression.java, LZ77Compression.java, HuffmanCompression.java and RLECompression.java class’s. Then the instances get the size of compressed file .

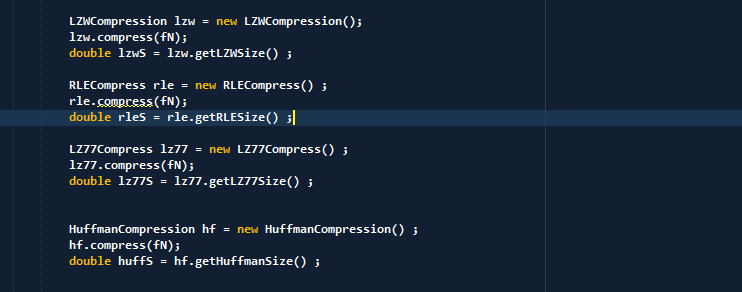


Figure: 2.4

After that Compression ratio is calculated dividing the compressed file size by original file size. Then compression ratio is shown by Bar Diagram using JFreeChart.

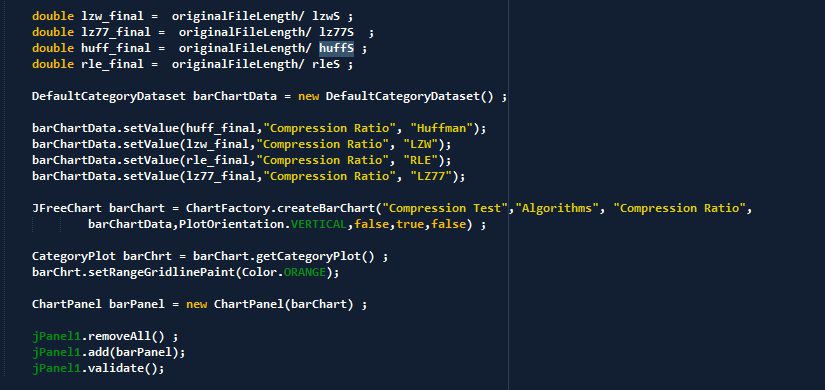


Figure: 2.5

After completing all the task the Interface will look like this.

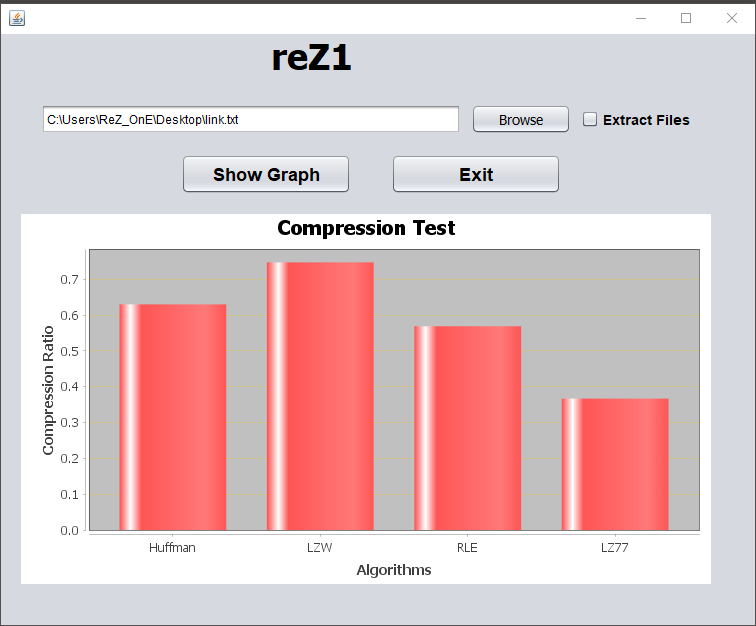


Figure: 2.6

3. **User Manual**

A frame will be visible after opening a jar file. Firstly the user have to select a .txt file from hard drive by clicking “browse” button. Then file manager will appear and selecting a file user have to click on “open” button and then that file is selected for compression. User can check the radio button for file extraction. Finally clicking on “show graph” button user can see the compression ratio of different compression algorithm.

**4. Conclusion**

In this project I have focus on Lossless compression algorithm. That’s why I have chosen those four algorithms. And I keep the file extraction option thus if user want to read the compressed file then user would be able to do that. Finally what I can say is using this tool using can get a concept of the compression ratio of some algorithms.

**References**

1. <https://users.cs.cf.ac.uk/Dave.Marshall/Multimedia/node214.html>

last visited: 27/5/2018

## Data Compression: The Complete Reference Book by David Salomon

## https://www.coursera.org/learn/algorithms-part2/lecture/lQ4b0/lzw-compression

## last visited:26/5/2018

1. Terry A. Welch, "A Technique for High Performance Data Compression", IEEE Computer, Vol. 17, No. 6, 1984, pp. 8-19.
2. <https://en.wikipedia.org/wiki/Data_compression_ratio>

last visited: 28/5/2018

1. GitHub link:

https://github.com/RezowanTalukder/SPL\_1\_0933/tree/master/spl1/reZ1