Data Compression Project

Text Compression using RLE Encoding,LZW Encoding and Huffman Coding

Student: Samir-Constantin Prejbeanu

Coordinator: Cristian Mihaescu PhD

Table of Contents:

1. Data Compression

In computer science and information theory, **data compression**, **source coding**, or **bit-rate reduction** involves encoding information using fewer bits than the original representation.

Compression can be either lossy or lossless.

Lossless compression reduces bits by identifying and eliminating statistical redudancy. No information is lost in lossless compression.

Lossy compression reduces bits by identifying unnecessary information and removing it. The process of reducing the size of a data file is popularly referred to as data compression, although its formal name is source coding (coding done at the source of the data before it is stored or transmitted).

Compression is useful because it helps reduce resource usage, such as data storage space or transmission capacity.

Because compressed data must be decompressed to use, this extra processing imposes computational or other costs through decompression; this situation is far from being a free lunch. Data compression is subject to a space-time complexity trade-off. For instance, a compression scheme for video may require expensive hardware for the video to be decompressed fast enough to be viewed as it is being decompressed, and the option to decompress the video in full before watching it may be inconvenient or require additional storage.

The design of data compression schemes involves trade-offs among various factors, including the degree of compression, the amount of distortion introduced (*e.g.*, when using lossy data compression), and the computational resources required to compress and uncompress the data.

2. Data Compression Algorithms

DCA can be either lossy or lossless.

Lossless data compression algorithms usually exploit statistical redudancy to represent data more concisely without losing information, so that the process is reversible. Lossless compression is possible because most real-world data has statistical redundancy. For example, an image may have areas of colour that do not change over several pixels; instead of coding "red pixel, red pixel, ..." the data may be encoded as "279 red pixels". This is a basic example of **run-length encoding (RLE)**; there are many schemes to reduce file size by eliminating redundancy.

Lossy data compression is the converse of **lossless data compression**. In these schemes, some loss of information is acceptable. Dropping nonessential detail from the data source can save storage space. Lossy data compression schemes are informed by research on how people perceive the data in question. For example, the human eye is more sensitive to subtle variations in luminance than it is to variations in color. JPEG image compression works in part by rounding off nonessential bits of information. There is a corresponding trade-off between preserving information and reducing size. A number of popular compression formats exploit these perceptual differences, including those used in music files, images, and video.

3. Huffman Coding

In computer science and information theory, a **Huffman Code** is an optimal prefix code found using the algorithm developed by David A. Huffman while he was a Ph.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes".

The process of finding and/or using such a code is called **Huffman Coding**, and is a common technique in entropy encoding, including lossless data compression. The algorithm's output can be viewed as a variable length code table for encoding a source symbol (such as a character in a file). Huffman's algorithm derives this table based on the estimated probability or frequency (*weight*) of occurrence for each possible value of the source symbol. As in other entropy encoding methods, more common symbols are generally represented using fewer bits than less common symbols. Huffman's method can be efficiently implemented, finding a code in linear time to the number of input weights if these weights are sorted.

However, although optimal among methods encoding symbols separately, Huffman coding is not always optimal among all compression methods.

3. Huffman Coding

Technique:

Given an input text, first of all there must be counted the frequency of each character in the text.

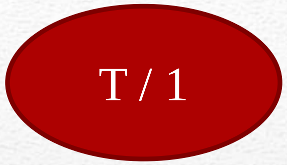
For example, if “The sea is wet.” would be the input, the following symbol – frequency table is generated:

“T”	**1**
“h”	1
“e”	3
“ “	3
“s”	2
“a”	1
“i”	1
“w”	1
“t”	**1**
“.”	1

3. Huffman Coding

Each line of the table represents a Huffman Tree Node.

Example:



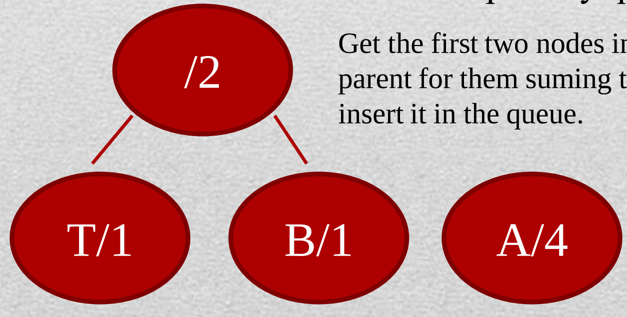
Where T is the symbol in the node and 1 is it's frequency in the text.

Nodes containing both a symbol and it's frequency represent the **leaf nodes** of the tree. Leaf nodes are kept in a **priority queue** used to build the tree.

Nodes with lower frequency are the first to be used, keeping characters with low frequency at the base of the tree and those with higher frequency closer to the root of the tree making sure we get a shorter **Huffman Code** for these, this resulting in a shorter output and a much better **Compression Ratio.**

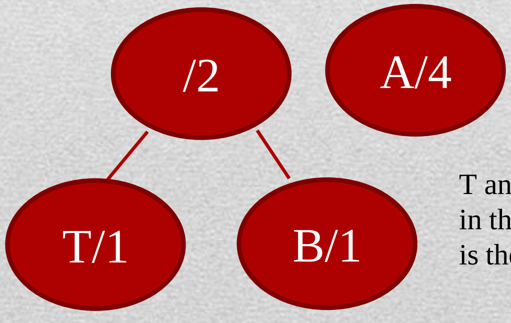
Creating internal nodes. Example:

Let's assume this is the head of our priority queue, an internal node is created like this:



Get the first two nodes in the queue, create a parent for them suming their frequencies, and insert it in the queue.

=>



T and B are no longer in the queue; [/2] [A/4] is the new queue.

3. Huffman Coding

A Huffman Code is generated given the position of a node in the tree.

Starting from the root each left branch is marked with “1” and each right branch is marked with “0”. That being said, a node who is left-right-right from the root has 100 as code.

Given our input we get the following symbol – code table:

“T”	**0101**
“h”	0100
“e”	10
“ “	000
“s”	110
“a”	0010
“i”	111
“w”	0110
“t”	**0111**
“.”	0011

These are variable-length prefix-free codes. Perfix-free means no confusion in the decoding process so no data is lost.

The length of each symbol is given by it’s position in the tree. It can be seen that characters with higher frequencies that are closer to the root have a shorter code.

3. Huffman Coding

Having the codes computed, the encoding process can be done resulting in the following output:

01010100100001101000100001111100000110100111001 1

where 0101 is “T”, 0100 is “h” etc. This is done by replacing each character in the input with it’s code.

The **decoding** works like this:

- Read the encoded sequence
- Starting from the root, every digit read means a step (0 to the right, 1 to the left)
- We walk down the tree until we find a symbol and output it
- Get back to the root
- Repeat until the decoding is done

4. LZW

Lempel–Ziv–Welch (**LZW**) is a universal lossless data compression algorithm created by Abraham Lempel, Jacob Ziv and Terry Welch. It was published by Welch in 1984 as an improved implementation of the LZ78 algorithm published by Lempel and Ziv in 1978.

The algorithm is simple to implement, and has the potential for very high throughput in hardware implementations. It was the algorithm of the widely used UNIX file compression utility compress, and is used in the GIF image format.

A high level view of the encoding algorithm is shown here:

- Initialize the dictionary to contain all strings of length one.
- Find the longest string W in the dictionary that matches the current input.
- Emit the dictionary index for W to output and remove W from the input.
- Add W followed by the next symbol in the input to the dictionary.
- Go to Step 2.

A dictionary is initialized to contain the single-character strings corresponding to all the possible input characters (and nothing else except the clear and stop codes if they're being used).

The algorithm works by scanning through the input string for successively longer substrings until it finds one that is not in the dictionary. When such a string is found, the index for the string without the last character (i.e., the longest substring that *is* in the dictionary) is retrieved from the dictionary and sent to output, and the new string (including the last character) is added to the dictionary with the next available code. The last input character is then used as the next starting point to scan for substrings.

In this way, successively longer strings are registered in the dictionary and made available for subsequent encoding as single output values. The algorithm works best on data with repeated patterns, so the initial parts of a message will see little compression. As the message grows, however, the compression ratio tends asymptotically to the maximum.

4. LZW

Technique:

Example:

Given the initial Dictionary (where the code of each symbol is it's position in the dictionary).

A	0
B	1
C	2

Given the following input: "ABBABa"

Because A is the first and the longest match in the dictionary it's code is printed, and A and the character next to it is added into the dictionary on a new position resulting in a new dictionary. Next time AB is found 3 is printed instead of printing 0 and 1 and ABa is added to the dictionary.

A	0
B	1
C	2
AB	3

5. RLE

Run-length encoding (**RLE**) is a very simple form of data compression in which *runs* of data (that is, sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run.

This is most useful on data that contains many such runs: for example, simple graphic images such as icons, line drawings, and animations. It is not useful with files that don't have many runs as it could greatly increase the file size.

RLE may also be used to refer to an early graphics file format supported by CompuServe for compressing black and white images, but was widely supplanted by their later GIF. RLE also refers to a little-used image format in Windows 3.x, with the extension **rle**, which is a Run Length Encoded Bitmap, used to compress the Windows 3.x startup screen.

Typical applications of this encoding are when the source information comprises long substrings of the same character or binary digit.

Run-length encoding performs lossless data compression and is well suited to palette-based bitmapped images such as computer incons. It does not work well at all on continuous-tone images such as photographs, although JPEG uses it quite effectively on the coefficients that remain after transforming and quantizing image blocks.

Common formats for run-length encoded data include Truevision TGA, PackBits, PCX and ILBM.

5. RLE

For example, consider a screen containing plain black text on a solid white background. There will be many long runs of white pixels in the blank space, and many short runs of black pixels within the text. Let us take a hypothetical single scan line, with B representing a black pixel and W representing white:

WWWWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWW
WWWWWWWWWWWWWWWWWWBWWWWWWWWWWWWWW

If we apply the run-length encoding (RLE) data compression algorithm to the above hypothetical scan line, we get the following:

12W1B12W3B24W1B14W

This is to be interpreted as twelve Ws, one B, twelve Ws, three Bs, etc.

6. Code Overview

Data Structures & Main Functions :

1. Huffman Coding

1.1 Data Structures

1.1.1 The Dictionary Structure

```
typedef struct Huffman
{
    string Symbol;                  // The symbol
    int Frequency;                  // It's frequency
    string Code;                    // It's computed code
};
```

1.1.2 The Tree Node

```
typedef struct HuffmanTree
{
        int Frequency;                          // Frequency
        string Symbol;                          // Symbol
        struct HuffmanTree *left, *right;       // Left Child & Right Child
};
```

6. Code Overview

1.2 Main Functions

void CountFrequencies(…)	// While reading the input, counting and storing every symbol’s frequency
HNode NewLeafNode(…)	// For every symbol a leaf node is created
HNode NewInternalNode(…)	// Creating internal nodes, suming the frequencies of it’s children
void QInsert(…)	// Every node created is inserted into the queue
HNode GetNode(…)	// In order to create internal nodes we must get nodes from the priority queue
void BuildTree(…)	// Builds the tree as a result of the functions above
void EncodeTree(…)	// Every left branch in the tree is marked with ‘1’ while every right branch is marked with ‘0’
string HuffmanCompress(…)	// Replacing every character in the input with it’s related code from the dictionary; the codes are computed in the EncodeTree() function
string DecodeText(…)	// For every digit read we walk down the tree(0 for right,1 for left, until we find a symbol and replace the code sequence with it

6. Code Overview

2. LZW

2.1 Data Structures

2.1.1 The Dictionary Structure

```
typedef struct LZW
{
    string Symbol;                    // The Symbol
    int Code;                         // It’s position in the Dictionary
};
```

2.2 Main Functions

void FillDictionaryLZW(…);	// At the start a dictionary is initialized containing 0-127 ASCII Characters
void AddToLZWDictionary(…);	// Every time a sequence not found in the dictionary is found it is added to the dictionary
string LZWCompress(…);	// While reading the text, find the longest substring S that is in the initial dictionary and output it’s code; add S and it’s following character to the dictionary on a new position

6. Code Overview

3. RLE

3.1 Data Structures

3.1.1 The Dictionary Structure

```
typedef struct RLE
{
    string Symbol;          //  The Symbol
    int SeqLen;             //  The length of the sequence the symbol repeats itself
                            until a different symbol occurs

};
```

3.2 Main Functions

```
string RLECompress(…);        // Given an input text, for every symbol met is computed
                              the length of the sequence he repeats itself until a different
                              symbol occurs ( Example: Anne = 1A2n1e).

string RLEDecompress(…);      // Given an encoded text, every pair LengthSymbol is
                              replaced with Length times Symbol ( Example: 1A2n1e =
                              Anne).
```

7. GUI

The **GUI** was created using Qt Creator.

It's a simple Graphic Interface containing only 4 buttons and 2 text browsers.

The text browsers are used to print the Input and the Output. They were added to the UI using Qt Creator's drag and drop.

As an alternative, their location and size can be computed in coding using the *setGeometry*(…) and *resize*(…) functions from the Qt Libraries.

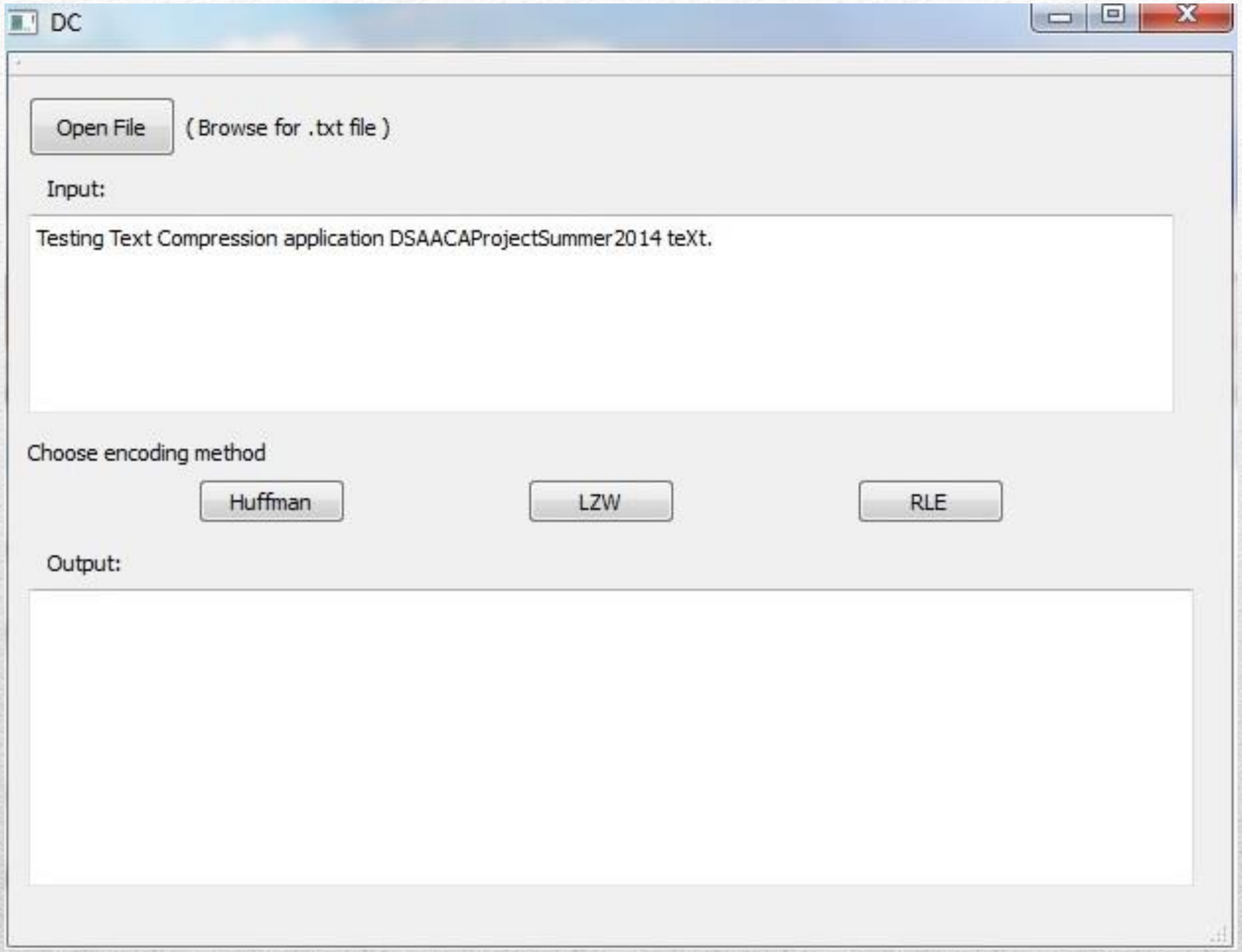
The 4 buttons are:

- Browse button // Opens a text file, loads it's content into a string and display it in a textBrowser
- Huffman button
- LZW button
- RLE button // On clicking each of these buttons, the correspoding encoding method is called and the encoded sequence is shown in the output textBrowser

The buttons also were added using Qt Creator's drag and drop.

8. Example

After browsing and loading a .txt file, the Input textBrowser shows the content of the file:

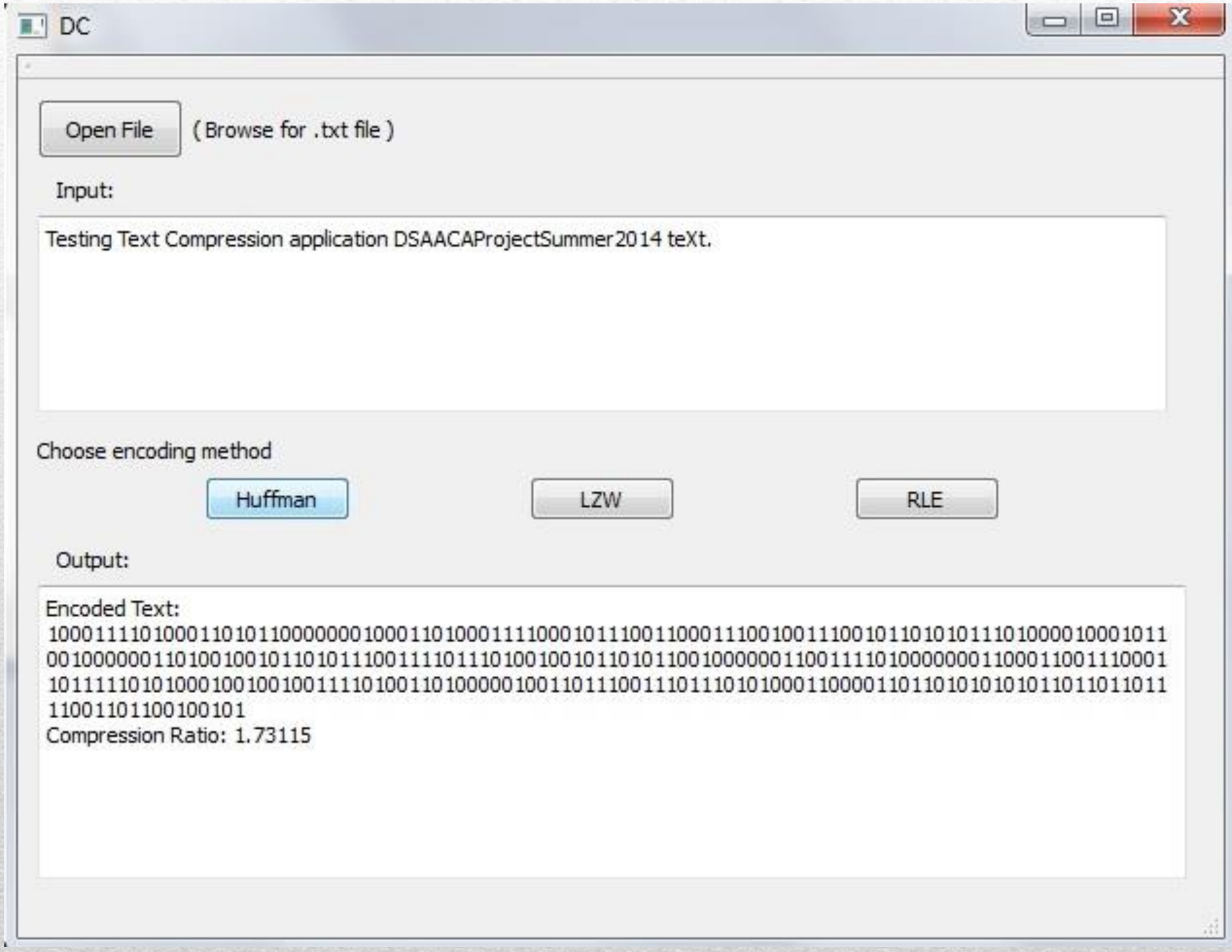


Testing Text Compression application DSAACAProjectSummer2014 teXt.

8. Example

Having the input stored in a string, after choosing the encoding method the encoded version of that string is shown in the output textBrowser.

Also, the compression ratio is computed and shown to the user:



```
Testing Text Compression application DSAACAProjectSummer2014 teXt.
```

```
Encoded Text:
1000111101000110101100000000100011010001111000101110011000111001001110010110101011101000010001011
001000000110100100101101011100111101110100100101101011001000000110011110100000001100011001110001
1011111010100010010010011110100110100000100110111001110111010100011000011011010101011011011011
11001101100100101
Compression Ratio: 1.73115
```

9. References

Links:
en.wikipedia.org
stackoverflow.com
www.youtube.com
http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/
http://mathworld.wolfram.com/
http://qt-project.org/

Books:
1. Burdescu,Dan Dumitru,Mihaescu,Marian Cristian,Algorithms and Data Structures,Academica Griefswald,2012.
2.Kernighan,W. Brian,Ritchie,Dennis M.,The C Programming Language 2nd edition.
3.Drozdek,Adam, Data Structures Algorithms in C++ Second edition.
4.Cormen,Thomas H.,Leiserson Charles E.,Rivest,Ronald ,Stein,Clifford,Introduction to Algorithms 3rd edition