**Zip-Compression**

History:

The format was created by Phil Katz in 1989 and was first implemented in PKZIP as replacement for the ARC format. It was made an open format which can be used and implemented by anyone.

It became very popular among PC users since it could minimize the disk-space needed and also store files together in an archive. This was also very efficient when transferring files from one computer to another using floppy or network.

Usage:

For lossless compression.

To pack files together in an archive which is easier to distribute.

Compression algorithms:

Several different compression algorithms are used, such as Shrunk, Reduced, Imploded, DEFLATE etc. The most common used is the latter.

DEFLATE algorithm:

The DEFLATE algorithm uses a variation of the LZ77 and static Huffman encoding. The dictionary is 32 Kbyte long and sliding and the look-ahead buffer is 258 bytes.

* LZ77:

|  |  |  |  |
| --- | --- | --- | --- |
| encoded data <-- | Dictionary (32Kbyte) | Look-ahead buffer (258bytes) | -->data to be encoded |

The LZ77 algorithm works by using the previously encoded data as the dictionary and then using a structure to refer to this data:

15-bytes: How far back the element found in the dictionary is from the start of the look-ahead buffer.

5-bytes: How many characters used from the found element in the dictionary.

8-bytes: The next token in the look-ahead buffer.

This is in total 28 bytes which is 20 bytes more than storing a normal character, but when words are repeated the algorithm will save space.

If the string “thresholding\_is\_thrilling” ( \_ indicates a space) were to be compressed using LZ77 it would do as:

|  |  |  |
| --- | --- | --- |
| **Dictionary** | **Look-ahead** |  |
|  | thresholding\_is\_thrilling | (0,0,"t") |
| t | hresholding\_is\_thrilling | (0,0,"h") |
| th | resholding\_is\_thrilling | (0,0,"r") |
| thr | esholding\_is\_thrilling | (0,0,"e") |
| thre | sholding\_is\_thrilling | (0,0,"s") |
| thres | holding\_is\_thrilling | (5,1,"o") |
| thresho | lding\_is\_thrilling | (0,0,"l") |
| threshol | ding\_is\_thrilling | (0,0,"d") |
| threshold | ing\_is\_thrilling | (0,0,"i") |
| thresholdi | ng\_is\_thrilling | (0,0,"n") |
| thresholdin | g\_is\_thrilling | (0,0,"g") |
| thresholding | \_is\_thrilling | (0,0,"\_") |
| thresholding\_ | is\_thrilling | (4,1,"s") |
| thresholding\_is | \_thrilling | (3,1,"t") |
| thresholding\_is\_t | hrilling | (16,2,"i") |
| thresholding\_is\_thri | lling | (13,1,"l") |
| thresholding\_is\_thrill | ing | (3,1,"n") |
| thresholding\_is\_thrillin | g | (13,1,EOS) |

(EOS = End of string)

The uncompressed size of the string is 200 bits and the compressed version is 504 bytes. This is simply because the string is very short. Therefore it cannot be compressed using LZ77. For a much longer string compression would in almost all cases be possible – unless the entropy is very high.

Another example will with a short string show how compression is possible. The string is “AAAAAAAABBBBBBBCCCCCC”:

|  |  |  |  |
| --- | --- | --- | --- |
| **Dictionary** | **Look-ahead** |  |  |
|  | AAAAAAAABBBBBBBCCCCCC |  | (0,0,"A") |
| A | AAAAAAABBBBBBBCCCCCC |  | (1,7,"B") |
| AAAAAAAAB | BBBBBBCCCCCC |  | (1,7,"C") |
| AAAAAAAABBBBBBBC | CCCCC |  | (1,5,"EOS) |

Here, the uncompressed size of the string is 168 bytes, where the compressed is only 96 bytes.

An important thing to notice is that in the first many iterations the token (0,0,”..”) is saved which does not provide very good compression, but as iterations grow so will the dictionary and it will be more likely to find a long match in the dictionary which provide good compression.

* Huffman coding:

The first part of the DEFLATE algorithm constructed a long string of bytes. The second part of the DEFLATE algorithm is the Huffman coding.

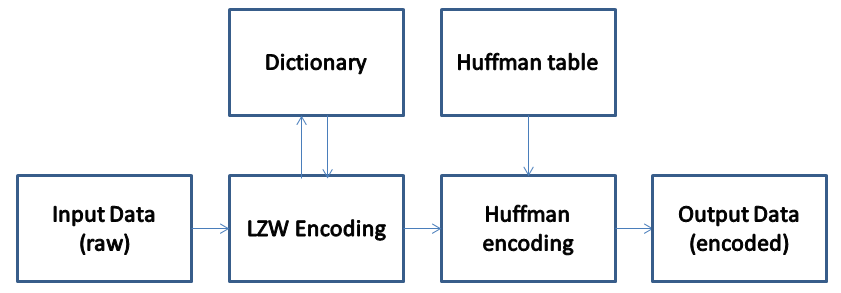
Huffman is used to encode sequences where certain compositions of bytes or chars occur more than others.

1. Look through the string and make a histogram of the possible compositions.
2. The higher frequency of the composition, the shorter the representing code should be.
3. Replace the string with the representation of the string in Huffman coding.

**Prototype**

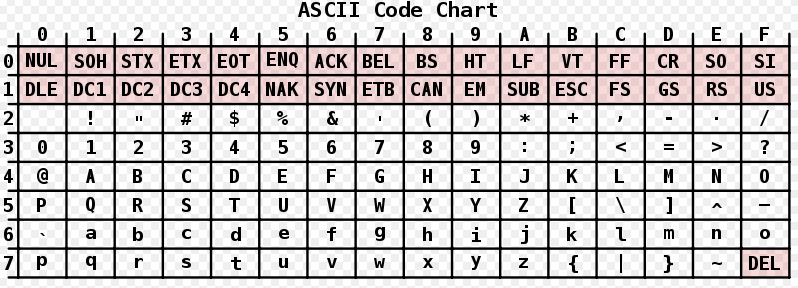
The prototype compression algorithm implemented is the LZW (Lempel-Ziv-Welch) which is a lossless data compression algorithm. It was created by Welch in 1983 as an improvement of the LZ78 algorithm. It is a fairly simple algorithm that relies on dictionaries for encoding and decoding.

The LZW became widely used when it became a part of the GIF image format I 1987 and it is still used in FreeBSD to this day.

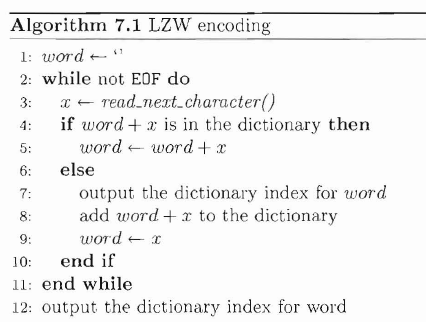
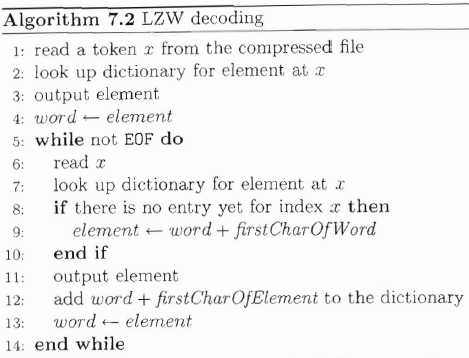
The process is:

LZW algorithm:

The dictionary used for compression and decompression is created during encoding and decoding, but there is a start dictionary. In this prototype this start dictionary consists of ASCII values from 0x20 to 0x7E:

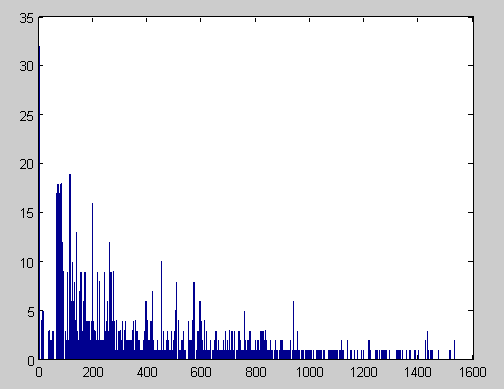


Encoding and decoding:

****The pseudo code for the encoding and decoding algorithm:

Huffman-coding:

Huffman coding in the prototype is done with the build in Huffman coding functions in MATLAB. These are very slow, but they do add some compression.



The above histogram shows the frequency of the different compositions of the encoded string. Normally this would be done on raw data (bytes), but this was done with the encoded LZW string directly.

Implementation:

The implementation of the LZW prototype was made in MATLAB which makes the readout of different registers and variables easy. It consists of the following files:

create\_dictionary.m Creates the start dictionary.  
decode.m Decodes LZW encoded ‘file’.   
decode\_string.m Finds the element in dictionary at x.  
encode.m Encodes LZW from ‘file’.  
encode\_string.m Finds the dictionary position for word.  
test.m Test-file with encoding -> decoding. Outputs compression ratio.  
test.txt Txt-file containing some Shakespeare chapter – 3.35KB.  
test2.txt Txt-file containing some text – is 18.9KB.

Test:

This test only considers compression ratio an important aspect. Speed is obviously not very good, because of the the implementation in MATLAB, no optimization and no hashing.

For the test.txt the compression ratio is 2.1. The compression ratio for Windows Zip (a standard application for many versions of Windows) generates a compression ratio of 2.22 . This compression ratio is probably smaller and better due to different compression in Windows Zip. This is still a bit odd since this also includes error correction checks, headers and other relevant information.

For the test2.txt the compression ratio is 2.21. For the Windows Zip it is 2.32.