Compression Session
CSC 461

Cyrus Erfani

Summer 2018

Table of Contents

Introduction

In the age of big data, compression is becoming more important every year. Fast and efficient compression techniques are required to store large amounts of data efficiently. There are different types of compression, which are necessitated by the existence of various forms of data. The different types of compression applied to certain data take advantage of the properties of that data. For example, one type of compression may utilize the fact that a stream of data remains unchanged within a file, while another may work based on the fact that the data changes very little.

In this report, three types of compression will be introduced. These are delta encoding, run-length encoding, and dictionary encoding. All of these compression techniques are lossless, meaning the data which is obtained after decompression is identical to the data prior to compression. In the following report, the theory behind each type of compression will be stated, and an implementation will be presented and evaluated.

Delta Encoding

Theory

Delta encoding is a synonym for change encoding. In delta encoding, the difference between consecutive blocks of data is stored rather than the actual data. Therefore, delta encoding takes advantage of the correlation between consecutive data to reduce the memory required to store the data. In order for this type of compression to be useful, it is required that correlation between consecutive data exist. If no such correlation exists, delta encoding will be unable to compress the data, as the difference will be too large to capture in fewer bytes than the original data.

Implementation

In this report, one possible implementation of delta encoding is presented[1]. In this implementation, encoding is done by taking a block of data and finding the numeric value of that block, which is then subtracted from the numeric value of the following block. The difference between the two is the value that is stored in the encoded file.

The means by which the difference is encoded depends on the magnitude of the difference. If the difference is small enough to fit within six bits, then one byte is used to encode the difference. The first six bits of the byte store the difference, and the seventh bit stores the sign of the difference; 1 for positive or 0 for negative difference. The eighth bit, which is the leading bit, is set to 1, which indicates that the current byte is the last byte of a number. In contrast, if a number is too large to fit within six bits, the trailing seven bits are iteratively right shifted out

[1] I created this implementation to illustrate delta encoding. Code is available on project site.

and padded with a 0 bit then stored, until the most significant 6 bits of the number can be stored in the final byte. The following example illustrates the process of encoding using this implementation.

Suppose that a block of data consists of 2 bytes. In a given file, the number value of one block is 5890 (00010111 00000010 in binary) and the value of the proceeding block is 6340 (00011000 11000100 in binary). The difference between the value of the blocks is:

$$6340 - 5890 = 450 \; (\, 00000001 \; 11000010 \; in \; binary \,)$$

To encode this difference, begin by observing that the number is too large to fit within six bits. Thus, we right shift the number by seven, to filter out the first seven bits. This is padded with a zero and stored in the encoded file. After right shifting, the remainder of the number can fit within six bits, so the it is padded with 11 and stored in the encoded file. To show this visually, colour-coded bits are shown below, which show where the bytes end up in the encoded file.

450 in binary:

00000001 11000010

After encoding:

. . . 11001001 01000010 11000011 11001011 . . .

As shown, the first seven bits (green) are right shifted out and padded with 0 (blue), then stored. After shifting, only bit sequence 000000011 remains. This binary number has a value of 3 in decimal, which is small enough to fit within six bits. So this remaining sequence is right shifted by six, and padded with 11. The first 1 (orange) indicates that 450 is a positive number, and the second 1 (blue) indicates that the byte is the last byte of the difference.

One advantage of this implementation is that no boundary byte is needed to separate the difference. Two unique numbers are discernable by which bytes have a leading 1. Another advantage is that the implementation is general in the sense that it will always work when there is a strong correlation between every block of data.

It must be noted that for this implementation to work, the first block of a file must be encoded as it is seen in the original file. This becomes the base number on which decoding is done. It is also worth noting that a block cannot be smaller than two bytes, because the difference cannot be stored in less than one byte.

Testing Methodology

To test this implementation, a special dataset was created in which each block of data is strongly correlated with the previous and following blocks. The dataset consisted of four text files full of numbers, where each number differed from the following and previous numbers by a relatively small amount. A sample line from this data set follows:

. . .
2111973.07,2111341.20,2111690.03,2111616.81,2111233.48,2111600.97,2111347.17,2111417.56,2111402.12,2111960.51 . . .

The test files consist of 100 000 samples each, and the correlation between the numbers can be seen. The numbers are in the millions, while the difference between any two consecutive numbers is at most in the thousands. Thus, for this particular set, if each digit is treated as one character (byte), then each block of data consists of 11 bytes (including decimal point and comma).

Each text file has a different block size, but the same average difference between the numbers in the set. This average difference is about 550. This will allow us to observe how the compression ratio will change when the correlation between the samples is changed.

Performance

The performance of the implementation is presented in Figure 1.

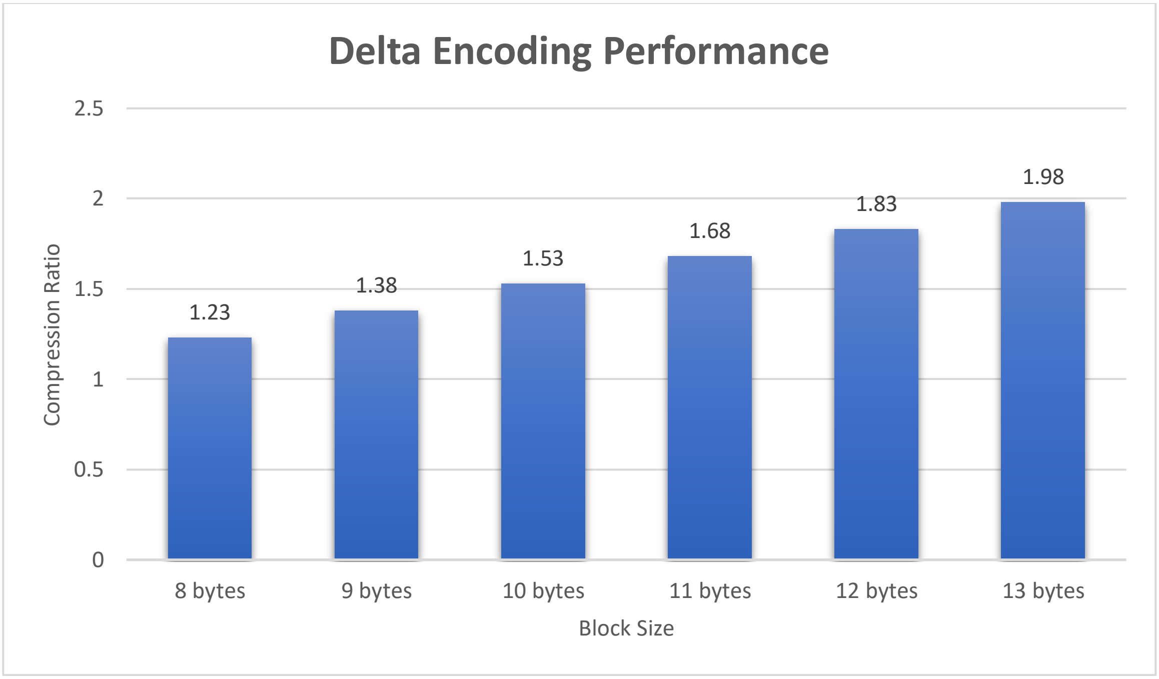


Figure 1

The data shows that the compression ratio improves under one circumstance: when the correlation between the data increases. This can be done in two ways: increasing the size of the data while keeping the average difference the same, or reducing the average difference between the consecutive samples.

Run Length Encoding

Theory

Run-length encoding (RLE) works by eliminating duplicate data. RLE takes advantage of the fact that in some circumstances, data remains stagnant, or does not change very often. For example, in JPEG compression where quantization is used, some data is quantized to the same value. In this case, RLE is used to compress the resulting image by encoding how many times a value occurred, rather than the value itself. There are various flavours of RLE but in this report three will be examined. All three have been implemented and the results of compressing files using each of these implementations to compress files will be presented. [2]

Implementation One

The first implementation is the most basic of any type. In this version, the number of times that a block of data repeats consecutively is stored. For example, suppose that in some file, we have the string shown below occurs.

A A A B C A C C C D D A A A A A

In this implementation, this string would be encoded as:

A 3 B 1 C 1 A 1 C 3 D 2 A 5

Simply stated, a count of how many times a character occurs follows the character itself. Thus, the first sequence of three As is followed by a 3, shown in red. Similarly, the count for all the other characters is also encoded, even when a character occurs only once.

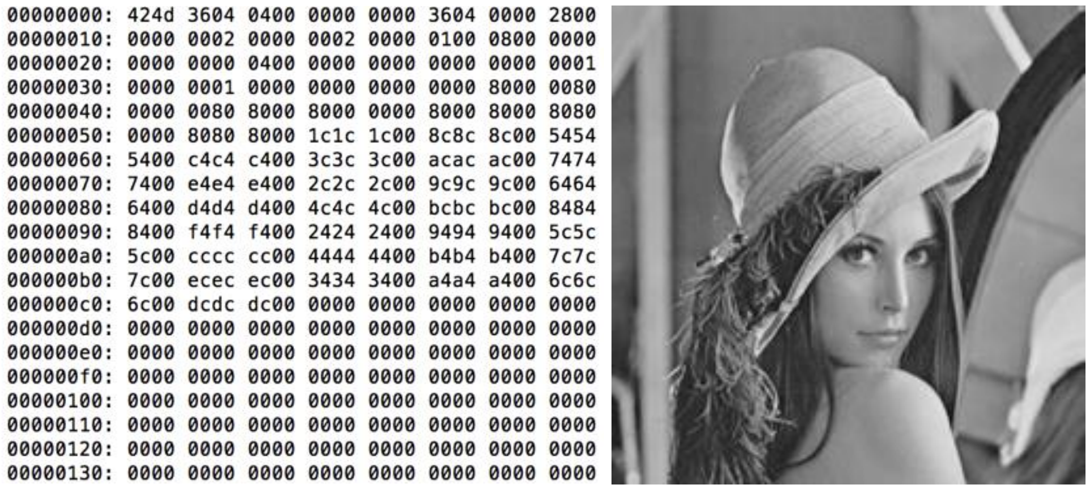
This implementation is impractical for most files, since streams of constant blocks rarely occur except in special circumstances like that of the JPEG compression algorithm stated above. In fact, applying this type of compression to most files will result in an encoded file which is larger than the original, as will be shown further into the report.

Implementation Two

The second implementation of RLE is smarter than the first because it will not substantially increase the size of a file if the data within contains no ‘runs’. In this version, only counts of the

[2] My code is available on project site

zero byte is encoded. This is due to the fact that zero bytes are likely to occur consecutively. This can be demonstrated by examining the raw data of a BMP image, as shown in Figure 2.



```
00000000: 424d 3604 0400 0000 0000 3604 0000 2800
00000010: 0000 0002 0000 0002 0000 0100 0800 0000
00000020: 0000 0000 0400 0000 0000 0000 0000 0001
00000030: 0000 0001 0000 0000 0000 0000 8000 0080
00000040: 0000 0080 8000 8000 0000 8000 8000 8080
00000050: 0000 8080 8000 1c1c 1c00 8c8c 8c00 5454
00000060: 5400 c4c4 c400 3c3c 3c00 acac ac00 7474
00000070: 7400 e4e4 e400 2c2c 2c00 9c9c 9c00 6464
00000080: 6400 d4d4 d400 4c4c 4c00 bcbc bc00 8484
00000090: 8400 f4f4 f400 2424 2400 9494 9400 5c5c
000000a0: 5c00 cccc cc00 4444 4400 b4b4 b400 7c7c
000000b0: 7c00 ecec ec00 3434 3400 a4a4 a400 6c6c
000000c0: 6c00 dcdc dc00 0000 0000 0000 0000 0000
000000d0: 0000 0000 0000 0000 0000 0000 0000 0000
000000e0: 0000 0000 0000 0000 0000 0000 0000 0000
000000f0: 0000 0000 0000 0000 0000 0000 0000 0000
00000100: 0000 0000 0000 0000 0000 0000 0000 0000
00000110: 0000 0000 0000 0000 0000 0000 0000 0000
00000120: 0000 0000 0000 0000 0000 0000 0000 0000
00000130: 0000 0000 0000 0000 0000 0000 0000 0000
```

Figure 2

In Figure 2, the image of Lena is show beside the raw data of her bmp image file. It can be seen that the zero byte occurs numerous times, consecutively, near the beginning of the file. This is also the case in other files where zero bytes occur in groups. This version of RLE aims to reduce file size by encoding the count of these zero bytes.

Let us revisit the example of a string of characters. Suppose the following string occurs within a file:

A A 0 0 0 B C A C 0 0 0 0 B D 0 0

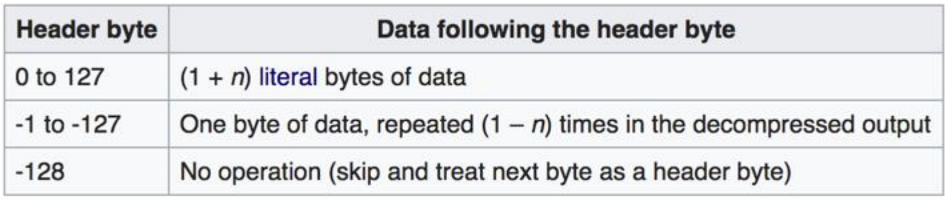
The string can be compressed by encoding the count of zeros as follows:

A A 0 3 B C A C 0 4 B D 0 2

Clearly, this reduces the size of the data, although not by much, as will be demonstrated.

Implementation Three

The final version of RLE examined will be Packbits [1]. Out of all the RLE schemes shown, Packbits is the 'smartest' one in the sense that it allows for sequences of changing data, as well as runs of constant data. This is accomplished by using a header byte which instruct the decoder how to treat the following bytes. The details of the header byte are shown in Figure 3.



Header byte	Data following the header byte
0 to 127	(1 + *n*) literal bytes of data
-1 to -127	One byte of data, repeated (1 – *n*) times in the decompressed output
-128	No operation (skip and treat next byte as a header byte)

Figure 3 [2]

In Packbits, the header byte is a signed byte. If the header byte has a value *n* in the range [0, 127], then the decoder will output *n+1* blocks of data following the header byte, exactly as it is shown in the encoded file. If the header byte has a value in the range [-1, -127], then the decoder will output *1-n* repetitions of the block following the header byte.

As an example, suppose the following sequence of bytes occur in a file:

```
AA AA AA 80 00 2A AA AA AA AA 80 00 2A 22 AA AA AA AA AA AA AA AA AA AA
```

The segments of this sequence would be encoded as:

```
FE AA                    ; (-(-2)+1) = 3 bytes of the pattern $AA
02 80 00 2A              ; (2)+1 = 3 bytes of discrete data
FD AA                    ; (-(-3)+1) = 4 bytes of the pattern $AA
03 80 00 2A 22           ; (3)+1 = 4 bytes of discrete data
F7 AA                    ; (-(-9)+1) = 10 bytes of the pattern $AA
```

The final output of the encoder for this string is:

```
FE AA 02 80 00 2A FD AA 03 80 00 2A 22 F7 AA
*     *           *     *              *
```

The bytes marked with a start are the header bytes. As apparent, the size of the input string was significantly reduced using Packbits.

Testing Methodology

In order to test the performance of these three variations of RLE, a substantial dataset was created. This dataset consisted of a spectrum of popular and well known file formats:

- bmp – 20 samples
- docx – 20 samples
- gif – 9 samples
- pdf – 18 samples
- tif – 17 samples
- wav – 1500 samples

The number of samples for each file type varies because the file types have different sizes. However, the total size of the samples in each file type was roughly the same. Some of these file formats were also selected because they are uncompressed, meaning they are good candidates for testing these compression algorithms. The RLE implementations were applied to these files in order to find how effective each version is. Note that for evaluation, the block size was set to one byte.

For the sources of the testing datasets, please see the references section of this report.

Performance

The implementations were testing using the dataset described above. The results of the testing are shown in Figure 4.

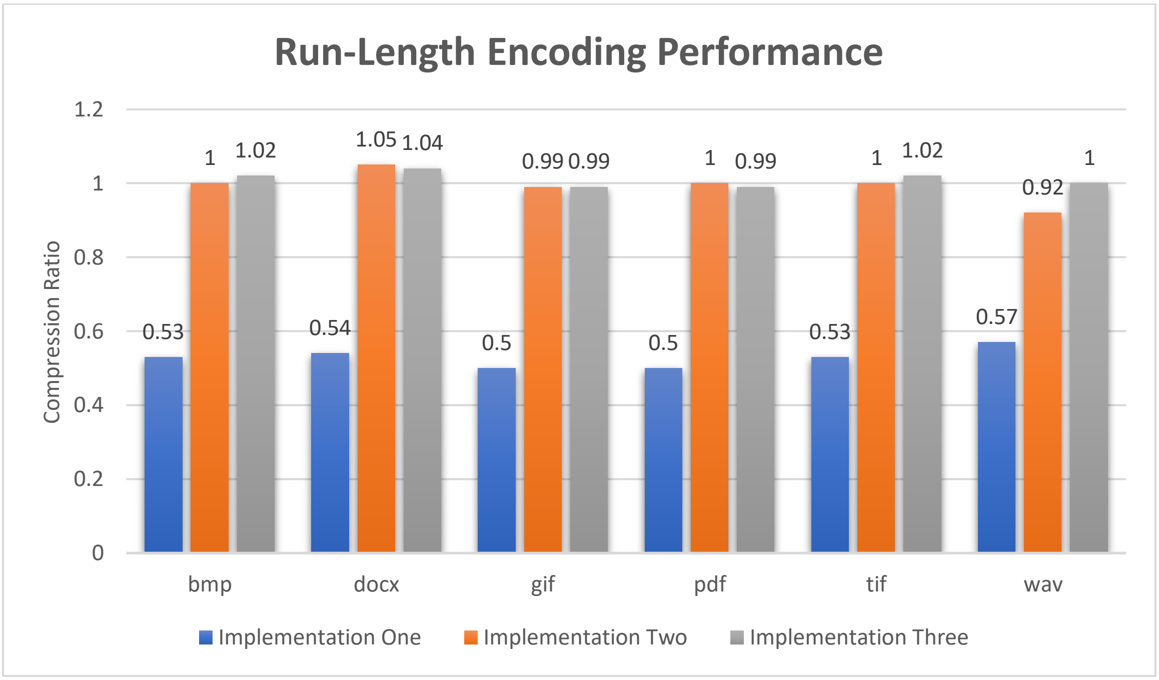


Figure 4

According to the results, Packbits is arguably the best choice of the three, having a compression ratio of one or greater in almost all circumstances. Implementation one is the absolute worst performing version, having nearly doubled the size of the data for every file type. This suggest there are very few occurrences of 'runs' of bytes in the files. This explains why all forms of RLE failed to significantly reduce the file sizes. Dictionary encoding may be better suited for these purposes.

Dictionary Encoding

Theory

Dictionary encoding works by using short ‘placeholder’ values in place of longer ‘words’. It does this by keeping a list of placeholders and their corresponding words. Therefore, dictionary encoding is good to use when sequences of bytes occur often than can be replaced by placeholder values. As an added benefit, it is possible to use a pre-built dictionary to encode a file. If the need arises, a dictionary can also be built from the file being encoded. These two types of dictionary encoding are called static and dynamic, respectively. In this report, only dynamic compression is considered.

Implementation

The implementation of dictionary encoding used in this report is Bytepairs, originally proposed by Phil Gage. [2] In Bytepairs, the idea is to iteratively replace a pair of bytes with one unused byte. It should be obvious that this cannot be done on the level of the entire file since all possible bytes will be used in a sufficiently large file. This is why the file must be split into blocks, and Bytepair compression conducted on the blocks. Regardless, a simplified example of Bytepairs is presented.

Suppose we have the following input string:

A A A B C A A A B D C

This string can be encode as through byte pairs as:

Z C Z D C
X = A A
Y = A B
Z = X Y

The dictionary for the encoding is shown below the final string. The way to decode the string is to start at the bottom of the dictionary and work up.

The implementation of byte pairs presented here was found online, can be found in the references.[2]

Testing Methodology

The testing methodology for byte pairs is identical to the testing methodology for RLE; refer to this section for methodology.

Performance

The performance of byte pairs is presented below, in Table

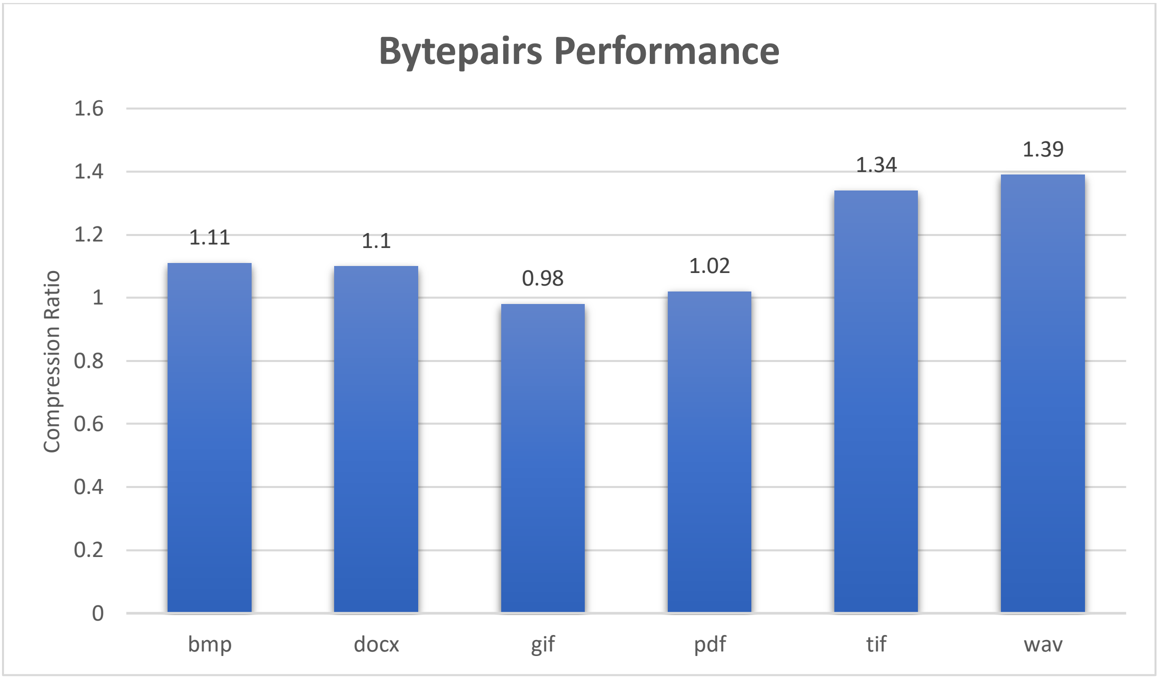


Figure 5

Bytepairs has proved to be the most successful of the compression schemes presented in this report. With the exception of gif and pdf, Bytepairs achieved reasonable amounts of compression, with WAV files experiencing the greatest compression. This suggests that WAV files have a lot of reoccurring pairs of bytes, as do TIF files. Interestingly, the results also seem to suggest that in contrast, filed like gif and pdf do not have many reoccurring pairs of bytes.

Conclusion

Delta encoding is effective under the circumstance that there is a strong correlation between consecutive blocks of data, else, the correlation is not enough to produce significant compression. Bytepairs works better, generally. It was show experimentally that Bytepairs produces alright results. Run-length encoding is special use and should not be applied generally.

References

[1]
https://web.archive.org/web/20080705155158/http://developer.apple.com/technotes/tn/tn1023.html

[2]
https://en.wikipedia.org/wiki/PackBits [table]

[3]
http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/CUJ/1994/9402/gage/gage.htm

Dataset References

Wav files:	https://github.com/Jakobovski/free-spoken-digit-dataset
Tif files:	http://www.imageprocessingplace.com/root_files_V3/image_databases.htm
Gif files:	https://giphy.com/
BMP files:	http://mmlab.ie.cuhk.edu.hk/projects/FSRCNN.html