Fixed Bit Coding

High-Speed, Lossless Data Compression for Small Data Sets

L. Stevan Leonard

July 24, 2020

# Introduction

Compressing small amounts of data is an application that most data compression algorithms cannot address. Most of these algorithms, including LZW methods, become viable with more than 32 data values. The fixed bit coding method described in this paper is designed to compress and decompress 2 to 64 bytes as quickly as possible. Many programmatic data sets are very small and would be suitable targets for data compression with an algorithm such as fixed bit coding. Possible applications of compression of small data sets include reducing memory storage requirements, such as an adjunct to a memory manager; and data compression on the fly, such as in embedded software during network data transfer.

Compression based on frequency of occurrence of data values is used by Huffman coding to produce optimal compression. But due to its data analysis, the required CPU execution time removes it as a high-speed application. Arithmetic coding, or range encoding, is another method that achieves high data compression at the expense of CPU time, which also removes that method from consideration, although it can be applied to small data sets. And due to the limited amount of data in 64 bytes, a dictionary lookup method is not applicable, although some algorithms, such as QuickLZ, can sometimes compress small data sets.

Rather than using frequency of occurrence, fixed bit coding improves on speed of execution over LZW, Huffman and arithmetic coding by encoding repeated values without analysis of frequencies. This approach has the drawback of somewhat lower data compression because the higher frequency values are not given fewer bits to encode them. Fixed bit coding achieves a good compression ratio by using a fixed number of bits to encode every input value. This results in extremely fast execution time for encoding and decoding. For highly compressible data, data with a lot of repeated values, fixed bit coding is initialized to generate at least 25% compression. Incompressible data is determined as early as possible so that the method runs as quickly as possible whether data is compressible or not.

# Overview

Fixed bit coding operates on data sets of 2 to 64 bytes that contain from 1 to 16 unique values. After the number of unique values in the data is found to be 16 or fewer, every value in the input is encoded with a 1- to 4- bit value that represents which of the unique values occurs at that position in the data. To be effective, the encoding must compress enough bits to offset the number of unique values. With these limits known ahead of time, decisions about compressibility can be made as early as possible (see Table 1).

Because the encoding uses the same number of bits for every input, the generation of output and the later decoding can be done efficiently, which reduces runtime. Unique values are written to the output array when they are first found reading the input, and the position of the corresponding unique value for each input value is stored.

The simplicity of this compression method leads to encoding and decoding at speeds of multiple 100s of megabytes per second and for 32 or more input values, decoding in excess of one gigabyte per second on a Mac with a 1 GHz Dual-Core Intel Core M processor. This method works only on data with a lot of repeated values and is initialized to compress only if the compression ratio is at least 25%. The implementation can be modified to achieve a lower or higher compression ratio, where possible.

# Implementation

Fixed bit coding is implemented in the files at <https://github.com/lsleonard/fixed-bit-coding>. The **fbc264** function compresses 2 to 64 values, and calls **fbc25** for two to five values. Call **fbc25** directly to avoid the call overhead. All functions are defined static and are included in the fbc.h header file. The execution of the fbc test bed requires an input file name, and optionally the block size (number of values to compress with default of 64) and loop count (with default of 1). The file of up to 20 Mbytes is read into memory. An array of unsigned long values is allocated to store whether data was compressed or not. This array and the block size are written to a file ending in .cq. The compressed or original data is written to a file ending in .fbc. The compressed percentage is printed, then compressed blocks, based on the bits returned by blocks that did compress, and compression time and rate. The decompress routine **fbc264d** or **fbc25d** is called after reading in the .cq and .fbc data files. Decompression rate and time is printed.

When the macro GEN\_STATS is defined, additional information about the data is printed, including number of uncompressed blocks and the percentage of encoded blocks by number of unique values.

As this algorithm is intended as a low-level tool for compression of small data sets, the implementation of how to manage compressed and uncompressed data is left for the application developer. For example, compressed data could be concatenated to save unused bits in the last byte of output. The test bed does not attempt to compress the bits that represent whether compression occurred or not, although this data could be highly compressed in some cases. The results from running the test bed are similar to what you can expect in a memory-based usage of the function, although the overhead of maintaining the file structure increases execution time, which negatively affects results as the number of input values decreases.

## Special Handling of 2 to 5 Values

Handling of 2 to 5 values is done in the functions **fbc25** and **fbc25d** to allow direct calls for these very small data sets. In addition to supporting output of one or two bytes for one unique value (all values the same), 2 and 3 values support compression when two nibbles define the bytes, and 4 or 5 values support 2 unique values.

## Determining Minimum Input Values for Number of Unique Values

At the core of fixed bit coding is the matching of number of unique values to number of input values. As the number of unique values in a data set increases, the compression ratio decreases because there are fewer repeated values that can be replaced by a shorter code. Table 1 lists the minimum input values for each number of unique values, the corresponding bits generated using the algorithm, and the corresponding compression savings with a target of 25%. As more input values occur for a given number of unique values, the resulting compression savings increase. The table used by the algorithm can be modified to achieve higher or lower compression ratios, where possible, by adjusting the minimum number of input values required for a given number of unique values.

Table 1: FBC Minimum Input Values for Number of Unique Values

|  |  |  |  |
| --- | --- | --- | --- |
| Number of Unique Values | Minimum Number of Input Values | Bits Generated | Data Compression Savings |
| 1 | 2 | 8, 10 | 50%, 37% |
| 2 | 4 | 25 | 22% |
| 3 | 7 | 44 | 21% |
| 4 | 9 | 56 | 22% |
| 5 | 15 | 90 | 25% |
| 6 | 17 | 104 | 23% |
| 7 | 20 | 121 | 24% |
| 8 | 23 | 138 | 25% |
| 9 | 40 | 240 | 25% |
| 10 | 44 | 264 | 25% |
| 11 | 48 | 288 | 25% |
| 12 | 52 | 312 | 25% |
| 13 | 56 | 336 | 25% |
| 14 | 60 | 360 | 25% |
| 15 | 62 | 376 | 25% |
| 16 | 64 | 392 | 24% |

The bits generated by the algorithm are listed in Table 2.

Table 2: Bits Generated by Fixed Bit Coding

|  |  |
| --- | --- |
| Encoding Element | Bits Generated |
| Indicator bits for 1 to 4 encoding bits | 1 bit=5, 2 bits=6, 3 bits=5, 4 bits=8 |
| Input values | Number values \* encoding bits for number of unique values |
| Unique values | Number unique values \* 8 |

Using the elements in Table 2, the computation of bits required to compress the input values is:

The encoding for a single unique value is indicated by setting the low bit of the first byte to 1 and copying the low-order 6 bits of the value to the upper 6 bits. If the value has 00 in the upper bits, the second bit of the first byte is set to 1. Otherwise, that second bit is set to 0 and the upper two bits of the value are copied into the low-order two bits of the second byte. The compression ratio achieved varies from 37% for 2 values to 96% for 64 values, when two bytes are used to encode the repeated value. When one byte is used to encode a single value with 00 in the upper bits, compression ranges from 50% for 2 values to 98% for 64 values.

This algorithm does not generate an encoding for uncompressed data. The decoder must be supplied the number of original values. Otherwise, all data required to decode the original data is contained in the encoded data described above.

# Comparison of Huffman and Fixed Bit Coding Methods

The following tables show compression ratio comparisons for Huffman and fixed bit coding methods with 64 input values. The bits compared were the count of bits generated to encode every value by each method. For both methods, the bits to encode unique values are the same, and the bits to encode indicators are assumed to be the same. Table 3 shows the worst-case compression for fixed bit versus Huffman coding when only 1 of the unique values repeats.

Table 3: Worst Case Compression for Fixed Bit Coding versus Huffman: 1 Unique Value Repeats, 64 Values

|  |  |  |  |
| --- | --- | --- | --- |
| Unique Values | Huffman (Percent) | FBC (Percent) | FBC vs Huffman (Percent) |
| 2 | 84.4 | 84.4 | 0.0 |
| 3 | 82.4 | 70.3 | -12.1 |
| 4 | 80.3 | 68.8 | -11.5 |
| 5 | 78.1 | 54.7 | -23.4 |
| 6 | 75.8 | 53.1 | -22.7 |
| 7 | 73.4 | 51.6 | -21.9 |
| 8 | 71.1 | 50.0 | -21.1 |
| 9 | 68.8 | 35.9 | -32.8 |
| 11 | 63.7 | 32.8 | -30.9 |
| 13 | 58.6 | 29.7 | -28.9 |
| 15 | 53.5 | 26.6 | -27.0 |
| 16 | 51.0 | 25.0 | -26.0 |

Table 4 shows the best-case compression of fixed bit versus Huffman coding where all unique values repeat equally. Huffman coding produces the biggest compression improvement over fixed bit coding when the number of unique values for the bits required is the least. For example, coding 5 unique values in 3 bits or 9 unique values in 4 bits.

Table 4: Best Case Compression for FBC vs Huffman: All Unique Values Repeat Equally, 64 Values

|  |  |  |  |
| --- | --- | --- | --- |
| Unique Values | Huffman (Percent) | FBC (Percent) | FBC vs Huffman (Percent) |
| 2 | 84.4 | 84.4 | 0.0 |
| 3 | 74.6 | 70.3 | -4.3 |
| 4 | 68.8 | 68.8 | 0.0 |
| 5 | 62.3 | 54.7 | -7.6 |
| 6 | 57.4 | 53.1 | -4.3 |
| 7 | 53.5 | 51.6 | -2.0 |
| 8 | 50.0 | 50.0 | 0.0 |
| 9 | 45.7 | 35.9 | -9.8 |
| 11 | 38.7 | 32.8 | -5.9 |
| 13 | 32.6 | 29.7 | -2.9 |
| 15 | 27.5 | 26.6 | -1.0 |
| 16 | 25.0 | 25.0 | 0.0 |

# Benchmark Data: Fixed Bit Coding and QuickLZ

Although fixed bit coding runs on small data sets, a test bed was configured to make calls with a specified block size over the values in a file to produce compression results across all blocks. For this test, 8, 32 and 64 blocks were tested. The fixed bit coding and QuickLZ 1.5.1 Beta 7 benchmark results are shown in Table 5 with megabytes per second to encode and decode, and compression percentage. The programs were run multiple times in a programmed loop of 100 on a Mac with a 1 GHz Dual-Core Intel Core M processor. The files were selected from the Squash Compression Benchmark at <https://quixdb.github.io/squash-benchmark/>.

Table 5: Benchmark Speed (Mbytes/second) and Compression (% of original): Fixed Bit Coding (FBC) and QuickLZ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| File | FBC 8  MBs/MBs/Cmp | FBC 32  MBs/MBs/Cmp | FBC 64  MBs/MBs/Cmp | QuickLZ  MBs/MBs/Cmp |
| fireworks.jpeg | 362/976/-1.52% | 848/3002/-.37% | 1025/3841/-.19% | 625/13677/-.01% |
| alice29.txt | 417/1134/-.47% | 476/3709/.05% | 294/4002/.24% | 180/244/45.1% |
| mr | 232/686/23.7% | 308/1879/27.5% | 278/1671/33.3% | 199/213/52.1% |
| paper-100k.pdf | 320/853/5.52% | 664/2757/2.23% | 673/3531/1.88% | 282/164/10.3% |
| ptt5 | 251/495/67.7% | 300/1056/69.8% | 305/1143/64.6% | 435/590/83.1% |
| sum | 246/735/7.80% | 301/1738/8.70% | 251/1593/9.71% | 190/261/51.8% |

The mostly incompressible file fireworks.jpeg shows the compression speed increasing from data sets of 8 values to 64 values. This results in large part to the overhead of setting up for compression and the management of multiple blocks. For FBC 64, there are 7/8 fewer bits than for FBC 8 to indicate whether the data compressed or not. By uncommenting the GEN\_STATS macro in the source code, details about the data are printed, including number of uncompressed blocks and the distribution of number of unique values in compressed blocks. The random data in much of the file means that compression is stopped quickly when the 3, 7, or 16 maximum unique values for 8, 32, or 64 input values was reached. For FBC, over 99% of blocks were incompressible. This helps produce the high decompression speed. The incredibly fast decompress for fireworks.jpeg by QuickLZ highlights that fixed bit coding is working on small blocks that must be handled individually rather than for an entire file. FBC decompresses 2 to 10 times faster than QuickLZ in most cases.

Comparing the compression speed for fireworks.jpeg and alice29.txt, you can see the slower compression times for alice29.txt for both fixed bit coding and QuickLZ. For fixed bit coding, as soon as the maximum supported unique values are exceeded, compression stops. Text has repeated values throughout, so it takes longer to exceed the limit of unique values, which causes compression failure to be longer than for random data where most values are unique.

In mr for FBC 64, the average number of unique values is 7.59, ranging from 2 to 16, Encoding 4 bits is most time-consuming, and 1/3 (FBC 8) to 1/2 (FBC 64) of blocks were compressed. For paper-100k.pdf, the compressed data has 2 or 3 unique values, which encode and decode quickly, and the incompressible data is random, which is processed quickly. ptt5 compresses 86% (FBC 8) to 89% (FBC 64) of blocks. These have 2 to 3 unique values, and result in high-speed compression.

Finally, sum compresses 23% (FBC 8) to 31% (FBC 64) of blocks. The average number of unique values for 64 input values is 12, and unique values are interspersed with zeros and lengthen compression time for incompressible blocks. The data in this file is compressible by 5.45% when using 2-byte blocks, and 9.62% when using 4-byte blocks. This shows that even very small data sets can sometimes be compressed.

# Conclusion

This paper has shown that fixed bit coding produces compression ratios that approach those of Huffman coding. To do efficient compression on small data sets, Huffman coding produces optimal results for benchmarking compression ratios. This means that the compression ratios achieved by fixed bit coding are dependent on the amount of repetition in the data and cannot be significantly improved upon without further analysis of the frequency of occurrence of data values or the use of a different method that uses additional CPU time for data analysis. With compression ratios bounded by the repetition in the data, fixed bit coding produces these results with minimum instructions to maximize compression and decompression speeds.

Small data sets will only be compressible when they contain many repeated values. To make compression viable for these small data sets, the approach taken by fixed bit encoding includes bypassing data that cannot be compressed by 25% to limit the time spent checking incompressible data. Along with the resulting fast compression and decompression speeds, the fixed bit coding algorithm is the only known candidate for applications that could benefit from high-speed compression of small data sets.