Fixed Bit Coding

High-Speed, Lossless Data Compression for Small Data Sets

L. Stevan Leonard

November 8, 2020

# Introduction

Compressing small amounts of data is an application that most data compression algorithms cannot address efficiently. 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. By comparing compression ratios, fixed bit coding is estimated overall to be within 10% of Huffman coding, the optimal frequency-based algorithm.

The basic method employed by fixed bit encoding is well known as using only the number of bits required to encode the number of unique characters in a data set. This algorithm performs this task as efficiently as possible, in part by ending the algorithm early if the predetermined number of unique values is exceeded, but primarily because the algorithm requires very little analysis.

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. Because of its support for small data sets, fixed bit coding can be applied to subsets of a data set where highly compressible bytes are known to exist. This would minimize the number of bytes requiring analysis and focus on the areas of likely benefit.

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 Huffman coding by encoding repeated values without analysis of frequencies. This approach has the drawback of somewhat lower data compression, estimated to be about 10% overall, 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 2).

Because the encoding uses the same number of bits for every input, the generation of output and the later decoding can be done extremely efficiently, which reduces runtime. Unique values are written to the output array when they are first read from 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 many 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 character values to compress with default of 64) and loop count (with default of 1). The specified 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. Output files are named based on the input file. This array and the block size are written to a file appended with fbc.cq. The compressed or original data is written to a file appended with .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 .fbc.cq and .fbc data files. Decompression rate and time is printed.

When the COMPRESS\_1\_PERCENT macro is defined in fbc.h, the unique value limit is set based on producing a minimum of 1% compression rather than the default of 25% compression. Because the number of unique values is limited to 16, the results from this macro can only differ from 25% for 34 or fewer input values.

When the macro GEN\_STATS is defined in main.c, 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. Also, the number of input values is not stored in the compressed data. 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 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.

## Bits Generated by the Algorithm

The bits generated by the algorithm are listed in Table 1.

Table : Bits Generated by Fixed Bit Coding

|  |  |
| --- | --- |
| Encoding Element | Bits Generated |
| Indicator bits for 1 to 4 encoding bits (based on number of unique values) | 1 bit=5, 2 bits=6, 3 bits=5, 4 bits=8  First bit: 1 for only one unique else 0  Next 4 bits: number of unique values  Next 3 bits: output values or padding bits |
| Input values | (number of values - 1) \* number of encoding bits |
| Unique values | number of unique values \* 8 |

Using the elements in Table 1, the computation of bits required to compress the input values is:

The first input value always matches the first unique value, so only this value is not encoded.

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 because this value is not stored in the compressed data. Otherwise, all data required to decode the original data is contained in the encoded data described above.

## 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 2 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 : FBC Minimum Input Values for Number of Unique Values to Achieve 25% Compression

|  |  |  |  |
| --- | --- | --- | --- |
| Number of Unique Values | Minimum Number of Input Values | Bits Generated | Data Compression Savings (Percent) |
| 1 | 2 | 8, 10 | 50, 37 |
| 2 | 4 | 24 | 25 |
| 3 | 7 | 42 | 25 |
| 4 | 9 | 54 | 25 |
| 5 | 14 | 84 | 25 |
| 6 | 17 | 101 | 25 |
| 7 | 20 | 118 | 26 |
| 8 | 22 | 132 | 25 |
| 9 | 38 | 228 | 25 |
| 10 | 42 | 252 | 25 |
| 11 | 46 | 276 | 25 |
| 12 | 50 | 300 | 25 |
| 13 | 54 | 324 | 25 |
| 14 | 58 | 348 | 25 |
| 15 | 62 | 372 | 25 |
| 16 | 64 | 388 | 24 |

An alternative mode of running the algorithm is to accept 1% or greater compression by setting the COMPRESS\_1\_PERCENT macro. The associated values for this mode are shown in Table 3. The resulting compression over a set of input values is maximized by using this method, although the runtime can be longer. These values do not change for 64 input values as any number of unique values up to 16 result in 25% compression.

Table : FBC Minimum Input Values for Number of Unique Values to Achieve 1% Compression

|  |  |  |  |
| --- | --- | --- | --- |
| Number of Unique Values | Minimum Number of Input Values | Bits Generated | Data Compression Savings (Percent) |
| 1 | 2 | 8, 10 | 50, 37 |
| 2 | 3 | 23 | 4.2 |
| 3 | 5 | 38 | 5.0 |
| 4 | 7 | 50 | 10.7 |
| 5 | 9 | 69 | 4.2 |
| 6 | 11 | 83 | 5.7 |
| 7 | 12 | 94 | 2.1 |
| 8 | 14 | 108 | 3.6 |
| 9 | 20 | 156 | 2.5 |
| 10 | 22 | 172 | 2.3 |
| 11 | 24 | 188 | 2.1 |
| 12 | 26 | 204 | 1.9 |
| 13 | 28 | 220 | 1.8 |
| 14 | 30 | 236 | 1.7 |
| 15 | 32 | 252 | 1.6 |
| 16 | 34 | 268 | 1.5 |

# 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. Table 4 shows the worst case for fixed bit coding, a single unique value repeating, while Table 5 shows the best case, all unique values repeating. Alternatively, these tables show the best and worst cases for Huffman coding. Table 6 shows an intermediate case where half the unique values repeat. Lastly, a real-world comparison is described using English language text.

The bits compared are the count of bits generated to encode every value by each method plus, for Huffman header, the frequency of each unique that is required to build the binary tree for decoding. For both methods, the number of indicator bits is assumed to be 5 bits as described above. The frequency of unique values for 64 input values can be listed from most frequent to least as the order of occurrence of unique values is not needed for decoding. This means the bits required for the next value will be at most what is required for the current value, so they can be determined dynamically.

Table 4 shows the worst-case compression for fixed bit versus Huffman coding when only 1 of the unique values repeats. The Huffman header will always be 6 bits for this case. Fixed bit coding produces 20% less compression than Huffman coding, on average.

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Unique Values | Huffman  Bits | Header Bits | FBC Bits | Huffman (Percent) | FBC (Percent) | FBC vs Huffman (Percent) |
| 2 | 64 | 6 | 63 | 82.2 | 83.6 | 1.4 |
| 3 | 66 | 6 | 126 | 80.3 | 69.7 | -10.5 |
| 4 | 69 | 6 | 126 | 78.1 | 68.2 | -10.0 |
| 5 | 72 | 6 | 189 | 76.0 | 54.3 | -21.7 |
| 6 | 76 | 6 | 189 | 73.6 | 52.7 | -20.9 |
| 7 | 80 | 6 | 189 | 71.3 | 51.2 | -20.1 |
| 8 | 84 | 6 | 189 | 68.9 | 49.6 | -19.3 |
| 9 | 88 | 6 | 252 | 66.6 | 35.7 | -30.9 |
| 11 | 98 | 6 | 252 | 61.5 | 32.6 | -28.9 |
| 13 | 108 | 6 | 252 | 56.4 | 29.5 | -27.0 |
| 15 | 118 | 6 | 252 | 51.4 | 26.4 | -25.0 |
| 16 | 123 | 6 | 252 | 48.8 | 24.8 | -24.0 |

Table 5 shows the best-case compression of fixed bit versus Huffman coding where all unique values repeat equally. The Huffman header requires 6 bits for the first value, and for the next value no more than the bits to encode the previous value or the remaining values. Also, the final unique value’s repeat can be imputed. In this case, Huffman coding exceeds fixed bit compression only when the number of unique values for the bits required is the smallest for that number of bits, such as for coding 3 unique values in 2 bits or 5 unique values in 3 bits. Fixed bit coding here produces 2.3% more compression than Huffman coding, on average.

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Unique Values | Huffman  Bits | Header Bits | FBC Bits | Huffman (Percent) | FBC (Percent) | FBC vs Huffman (Percent) |
| 2 | 64 | 6 | 63 | 83.2 | 84.6 | 1.4 |
| 3 | 106 | 11 | 126 | 72.5 | 70.7 | -1.8 |
| 4 | 128 | 14 | 126 | 66.0 | 69.1 | 3.1 |
| 5 | 153 | 18 | 189 | 58.8 | 55.3 | -3.5 |
| 6 | 170 | 22 | 189 | 53.1 | 53.7 | 0.6 |
| 7 | 182 | 26 | 189 | 48.4 | 52.1 | 3.7 |
| 8 | 192 | 24 | 189 | 45.3 | 50.6 | 5.3 |
| 9 | 206 | 27 | 252 | 40.4 | 36.7 | -3.7 |
| 11 | 226 | 33 | 252 | 32.2 | 33.6 | 1.4 |
| 13 | 241 | 39 | 252 | 25.0 | 30.5 | 5.5 |
| 15 | 251 | 45 | 252 | 18.8 | 27.3 | 8.6 |
| 16 | 256 | 34 | 252 | 18.4 | 25.8 | 7.4 |

Table 6 shows that for 64 input values, when half the unique values repeat equally, FBC produces 4% less compression than Huffman coding, on average.

Table : FBC Versus Huffman with Half the Unique Values Repeating Equally, 64 Input Values

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Unique Values | Huffman  Bits | Header Bits | FBC Bits | Huffman (Percent) | FBC (Percent) | FBC vs Huffman (Percent) |
| 2 | 64 | 6 | 63 | 83.2 | 84.6 | 1.4 |
| 3 | 96 | 11 | 126 | 74.4 | 70.7 | -3.7 |
| 4 | 99 | 11 | 126 | 72.3 | 69.1 | -3.1 |
| 5 | 130 | 16 | 189 | 63.7 | 55.3 | -8.4 |
| 6 | 133 | 16 | 189 | 61.5 | 53.7 | -7.8 |
| 7 | 149 | 18 | 189 | 56.4 | 52.1 | -4.3 |
| 8 | 155 | 18 | 189 | 53.7 | 50.6 | -3.1 |
| 9 | 176 | 22 | 252 | 47.3 | 36.7 | -10.5 |
| 11 | 194 | 26 | 252 | 39.8 | 33.6 | -6.3 |
| 13 | 204 | 24 | 252 | 35.2 | 30.5 | -4.7 |
| 15 | 226 | 27 | 252 | 27.1 | 27.3 | 0.2 |
| 16 | 230 | 27 | 252 | 24.8 | 25.8 | 1.0 |

Huffman coding of English language text for 64 values produces about 50% compression, before subtracting header and unique values. For the file alice29.txt (see benchmark data below), 64 input values require an average of 23.7 unique values (190 bits) with 13.5 repeating values (about 40 header bits) and results in about 5% compression. Fixed bit coding yields less than 1% compression. This example points out that for small data sets, Huffman coding, though optimal, does not produce the high compression rate that is claimed for examples where the Huffman tree overhead is canceled out by larger amounts of data and can yield greater than 40% compression.

When considering the spectrum of compressible data for 64 input values, fixed bit coding should be within 10% of the compressed result of using Huffman coding, on average.

# 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 in that file. 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 7 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 : 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 | 434/1089/-1.52% | 898/3156/-.37% | 1256/4103/-.19% | 625/13677/-.01% |
| alice29.  txt | 417/1134/-.47% | 476/3709/.05% | 294/4002/.24% | 180/244/45.1% |
| mr | 232/686/23.7% | 310/1951/27.5% | 279/1671/33.3% | 199/213/52.1% |
| paper-100k.pdf | 354/957/5.52% | 706/2925/2.23% | 721/3938/1.88% | 282/164/10.3% |
| ptt5 | 266/562/67.7% | 300/1098/69.8% | 309/1182/64.6% | 435/590/83.1% |
| sum | 253/735/7.80% | 310/1738/8.70% | 258/1593/9.71% | 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.

# A Possible Algorithmic Option

An option for this algorithm was studied that applies the encoding to the number of repeated unique values rather than all unique values. Improved compression results when one (or more) fewer bits are required to encode the data. This method requires a tradeoff of the highest fixed bit value to represent unique values. For example, if a data set had 12 unique values and 6 of those repeated, then the encoded values 0 through 5 would be used for the repeating values and 7 for unique values. For 9 to 16 unique values to benefit from this method, 7 or fewer of those unique values can be repeating. For the files from benchmarking above, compression based on compressed blocks varied from a low of -0.71% (fireworks.jpeg) to a high of 17% (sum). Although runtime decreased very slightly for fireworks.jpeg and alice29.txt, runtime increased for other files. The highest increase in runtime of 29% was observed for an 8% increase in compressed blocks for ptt5. Although this option can increase compression, the tradeoff in runtime would have to be weighed against the type of data and application requirements. The code used for this option is not published.

# Conclusion

This paper has shown that fixed bit coding produces compression ratios that approach those of Huffman coding. Overall, fixed bit coding should be within 10% of Huffman coding. To make compression viable for small data sets, fixed bit coding bypasses data that cannot be compressed by 25% to limit the time spent checking incompressible data. Along with the resulting extremely 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.