

Basic Compression Library

Manual

API version 1.1
December 25, 2004

©2003-2004 Marcus Geelnard

Summary

This document describes the algorithms used in the Basic Compression Library, and how to use the library interface functions.

Contents

| | | |
|----------|-----------------------------------|-----------|
| 1 | Introduction | 1 |
| 1.1 | Background | 1 |
| 1.2 | The library philosophy | 1 |
| 2 | The Compression Algorithms | 2 |
| 2.1 | RLE | 2 |
| 2.1.1 | Principle | 2 |
| 2.1.2 | Implementation | 2 |
| 2.2 | Huffman | 3 |
| 2.2.1 | Principle | 3 |
| 2.2.2 | Implementation | 4 |
| 2.3 | Rice | 4 |
| 2.3.1 | Principle | 4 |
| 2.3.2 | Implementation | 6 |
| 2.4 | Lempel-Ziv (LZ77) | 7 |
| 2.4.1 | Principle | 7 |
| 2.4.2 | Implementation | 7 |
| 3 | Compiling | 9 |
| 4 | Library API Reference | 10 |
| 4.1 | RLE_Compress | 10 |
| 4.2 | RLE_Uncompress | 10 |
| 4.3 | Huffman_Compress | 10 |
| 4.4 | Huffman_Uncompress | 11 |
| 4.5 | Rice_Compress | 11 |
| 4.6 | Rice_Uncompress | 11 |
| 4.7 | LZ_Compress | 12 |
| 4.8 | LZ_CompressFast | 12 |
| 4.9 | LZ_Uncompress | 12 |
| 5 | License | 13 |

Chapter 1

Introduction

1.1 Background

This is a library of well known compression algorithms. They are not intended to serve as competitors to advanced general purpose compression tools, but rather as building blocks for specialized compression algorithms. A typical example is the compression of digitized media (such as images or audio), for which you usually have lots of apriori information, and can tailor an efficient method based on entropy reduction (e.g. differentiation) followed by simple entropy coding (e.g. Huffman coding).

In many cases entropy reduction results in data that is NOT easily compressed with advanced dictionary based entropy coders, such as gzip or bzip2, since the data often becomes very noisy. Even a simple Huffman coder can usually give better compression results than the most advanced dictionary based coders under these circumstances.

1.2 The library philosophy

All compression algorithms are represented by a coder and a decoder (a compressor and a decompressor), often referred to as a CODEC.

All coders and decoders work on preallocated memory buffers, and do not rely on any internal memory allocation or file I/O. In addition, all library functions are 100% reentrant.

No data integrity checking is performed within the library (e.g. there are no CRCs stored in the compressed data streams, nor any information about the size of the uncompressed data). This kind of functionality is left to the user of the library.

The entire library is written in portable ANSI C code. The code should work identically on 32-bit and 64-bit architectures, and on both little endian and big endian systems (with the exception that the Rice coder/decoder imports/exports data in machine native endian format).

Chapter 2

The Compression Algorithms

This chapter briefly explains each compression algorithm, as it has been implemented in the Basic Compression Library. For more in depth technical information, see the source code and a decent resource on compression algorithms.

2.1 RLE

RLE, or Run Length Encoding, is a very simple method for lossless compression. It simply replaces repeated bytes with a short description of which byte to repeat, and how many times to repeat it. Though simple and obviously very inefficient for general purpose compression, it can be very useful at times (it is used in JPEG compression, for instance).

2.1.1 Principle

An example of how a run length encoding algorithm can encode a data stream is shown in figure 2.1, where six occurrences of the symbol '93' have been replaced with three bytes: a marker byte ('0' in this case), the repeat count ('6'), and the symbol itself ('93').

When the RLE decoder encounters the symbol '0', which is used as the marker byte, it will use the following two bytes to determine which symbol to output and how many times to repeat the symbol.

2.1.2 Implementation

There are several different ways to do RLE. The particular method implemented in the Basic Compression Library is a very efficient one. Instead of coding runs for both repeating and non-repeating sections, a special marker byte is used to indicate the start of a repeating section.

| | | | | | | | | | | | |
|----|----|----|----|----|----|----|----|----|----|----|----|
| 12 | 65 | 14 | 52 | 53 | 93 | 93 | 93 | 93 | 93 | 93 | 32 |
|----|----|----|----|----|----|----|----|----|----|----|----|

...is encoded as...

| | | | | | | | | |
|----|----|----|----|----|---|---|----|----|
| 12 | 65 | 14 | 52 | 53 | 0 | 6 | 93 | 32 |
|----|----|----|----|----|---|---|----|----|

Figure 2.1: The principle of run length encoding.

Non-repeating sections can thus have any length without being interrupted by control bytes, except for the rare case when the special marker byte appears in the non-repeating section (which is coded with at most two bytes). For optimal efficiency, the marker byte is chosen as the least frequent (perhaps even non-existent) symbol in the input stream.

Repeating runs can be as long as 32768 bytes. Runs shorter than 129 bytes require three bytes for coding (marker + count + symbol), whereas runs longer than 128 bytes require four bytes for coding (marker + $\text{count} \gg 0x80$ + $\text{count} \ll 0x80$ + symbol). This is normally a win in compression, and it is very seldom a loss of compression ratio compared to using a fixed coding of three bytes (which allows coding a run of 256 bytes in just three bytes).

With this scheme, the worst case compression result is: $outsized = \frac{257}{256} \times insize + 1$.

2.2 Huffman

Huffman encoding is one of the best methods for lossless compression. It replaces each symbol with an alternate binary representation, whose length is determined by the frequency of the particular symbol. Common symbols are represented by few bits, while uncommon symbols are represented by many bits.

The Huffman algorithm is optimal in the sense that changing any of the binary codings of any of the symbols will result in a less compact representation. However, it does not deal with the ordering or repetition of symbols or sequences of symbols.

2.2.1 Principle

I will not go into all the practical details about Huffman coding, but the basic principle is to find new binary representations for each symbol so that common symbols use few bits per symbol, while uncommon symbols use more bits per symbol.

The solution to this problem is, in short, to make a histogram of the uncompressed data stream in order to find how common each symbol is. A binary tree is then created by recursively splitting this histogram in halves, where each half in each recursion should weigh as much as the other half (the weight is $\sum_{k=1}^N symbolcount_k$, where N is the number of symbols in the branch and $symbolcount_k$ is the number of occurrences of symbol k).

This tree serves two purposes:

1. The coder uses the tree to find the optimal representations for each symbol.
2. The decoder uses the tree to uniquely identify the start and stop of each code in the compressed data stream: by traversing the tree from top to bottom while reading the compressed data bits, selecting branches based on each individual bit in the data stream, the decoder knows that a complete code has been read once a leaf node is reached.

Let us have a look at an example to make it more clear. Figure 2.2 shows an uncompressed data stream consisting of ten bytes.

Based on the symbol frequencies, the Huffman coder comes up with the Huffman tree (figure 2.4) and the accompanying coded representation (figure 2.3).

As you can see, common symbols are closer to the root node (the top of the figure), thus requiring fewer bits for representation. Based on the found Huffman tree, the coder then encodes the data stream with the alternate representation, as can be seen in 2.5.

The total size of the compressed data stream is 24 bits (three bytes), compared to the original 80 bits (10 bytes). Of course, we would have to store the actual Huffman tree too so that the decoder is able to decode the compressed stream, which would probably make the true size of the compressed data stream *larger* than the input stream in this case. This is a side effect of the relatively short data set used in this example. For larger data sets, the overhead of the Huffman tree becomes negligible.

As an exercise, try decoding the compressed data stream by traversing the Huffman tree from top to bottom, selecting left/right branches for each new bit in the compressed data stream. Each time a leaf node is encountered, the corresponding byte is written to the decompressed output stream, and the tree traversal starts over from the root again.

2.2.2 Implementation

The Huffman coder that can be found in the Basic Compression Library is a very straight forward implementation.

Primary flaws with this primitive implementation are:

- Slow bit stream implementation
- Fairly slow decoding (slower than encoding)
- Maximum tree depth of 32 (the coder aborts if any code exceeds a size of 32 bits). If I am not mistaking, this should not be possible unless the input buffer is larger than 2^{32} bytes, which is not supported by the coder anyway (max $2^{32} - 1$ bytes can be specified with an unsigned 32-bit integer).

On the other hand, there are a few advantages of this implementation:

- The Huffman tree is stored in a very compact form, requiring only 12 bits per symbol (for 8 bit symbols), meaning a maximum of 384 bytes overhead.
- The code should be fairly easy to follow.

The Huffman coder does quite well in situations where the data is noisy, in which case most dictionary based coders run into problems.

2.3 Rice

For data consisting of large words (e.g. 16 or 32 bits) and mostly low data values, Rice coding can be very successful at achieving a good compression ratio. This kind of data is typically audio or high dynamic range images that has been pre-processed with some kind of prediction (such as delta to neighboring samples).

Although Huffman coding should be optimal for this kind of data, it is not a very suitable method due to several reasons (for instance, a 32-bit word size would require a 16 GB histogram buffer to encode the Huffman tree). Therefore a more dynamic approach is more appropriate for data that consists of large words.

2.3.1 Principle

The basic idea behind Rice coding is to store as many words as possible with less bits than in the original representation (just as with Huffman coding). In fact, one can think of the Rice code as a fixed

| | | | | | | | | | |
|----|----|----|----|----|----|----|----|----|----|
| 32 | 22 | 22 | 43 | 49 | 22 | 22 | 17 | 48 | 43 |
|----|----|----|----|----|----|----|----|----|----|

Figure 2.2: Uncompressed data stream, used as input to the Huffman coder.

| Symbol | Frequency | Code |
|--------|-----------|------|
| 22 | 4 | 00 |
| 43 | 2 | 01 |
| 17 | 1 | 100 |
| 32 | 1 | 101 |
| 48 | 1 | 110 |
| 49 | 1 | 111 |

Figure 2.3: Symbol frequencies and encoding for the data stream in figure 2.2.

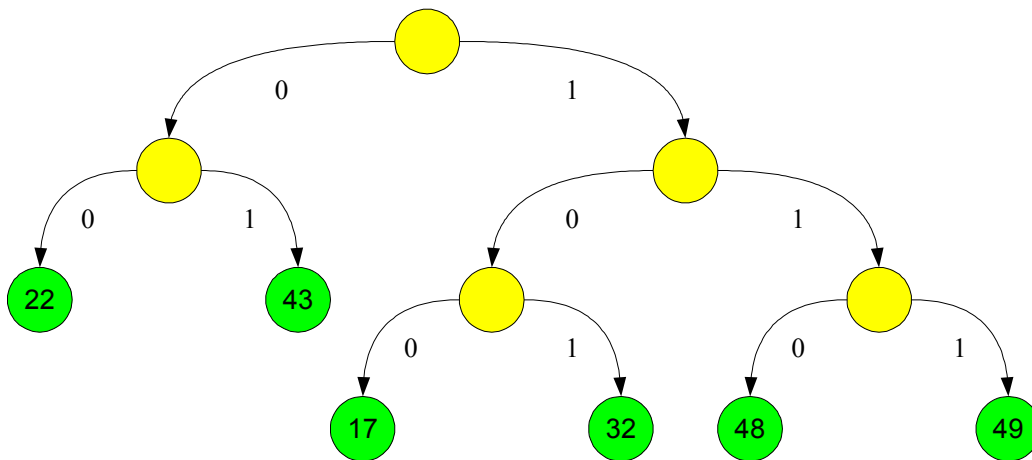


Figure 2.4: Huffman tree for the data stream in figure 2.2.

| | | | | | | | | | |
|-----|----|----|----|-----|----|----|-----|-----|----|
| 101 | 00 | 00 | 01 | 111 | 00 | 00 | 100 | 110 | 01 |
|-----|----|----|----|-----|----|----|-----|-----|----|

Figure 2.5: Compressed data stream.

Huffman code (i.e. the codes are not determined by the actual statistical content of the data, but by the assumption that lower values are more common than higher values).

The coding is very simple: Encode the value X with X '1' bits followed by a '0' bit.

2.3.2 Implementation

There are some optimizations in the Rice implementation that can be found in the Basic Compression Library:

1. The k least significant bits of each word are stored as is, and the N-k most significant bits are encoded with Rice coding. k is chosen as the average number of bits for the previous few samples in the stream. This usually makes the best use of the Rice coding, "hides" noise from the Rice coder, and does not result in very long Rice codes for signals with varying dynamic range.
2. If the rice code becomes longer than a fixed threshold, T, an alternate coding is used: output T '1' bits, followed by $\text{floor}(\log_2(X-T))$ '1' bits, and one '0' bit, followed by X-T (represented by the least significant $\text{floor}(\log_2(X-T))-1$ bits). This gives pretty efficient coding even for large values, and prevents ridiculously long Rice codes (in the worst case scenario, a single Rice code for a 32-bit word may become as long as 2^{32} bits, or 512 MB).

If the threshold is set to 4, then the following is the resulting code table:

| X | bin | Rice | Thresholded Rice | Difference |
|----|-------|-----------------------|------------------|------------|
| 0 | 00000 | 0 | 0 | |
| 1 | 00001 | 10 | 10 | |
| 2 | 00010 | 110 | 110 | |
| 3 | 00011 | 1110 | 1110 | |
| 4 | 00100 | 11110 | 11110 | |
| 5 | 00101 | 111110 | 111110 | |
| 6 | 00110 | 1111110 | 11111100 | +1 |
| 7 | 00111 | 11111110 | 11111101 | |
| 8 | 01000 | 111111110 | 1111111000 | +1 |
| 9 | 01001 | 1111111110 | 1111111001 | |
| 10 | 01010 | 11111111110 | 1111111010 | -1 |
| 11 | 01011 | 111111111110 | 1111111011 | -2 |
| 12 | 01100 | 1111111111110 | 111111110000 | |
| 13 | 01101 | 11111111111110 | 111111110001 | -1 |
| 14 | 01110 | 111111111111110 | 111111110010 | -2 |
| 15 | 01111 | 1111111111111110 | 111111110011 | -3 |
| 16 | 10000 | 11111111111111110 | 111111110100 | -4 |
| 17 | 10001 | 111111111111111110 | 111111110101 | -5 |
| 18 | 10010 | 1111111111111111110 | 111111110110 | -6 |
| 19 | 10011 | 11111111111111111110 | 111111110111 | -7 |
| 20 | 10100 | 111111111111111111110 | 11111111100000 | -5 |

As you can see, only two codes result in a worse representation with the threshold method used in this implementation. The rest of the codes result in shorter or equally short codes as for standard Rice coding.

3. In the worst case scenario, the output buffer may grow by several orders of magnitude compared to the input buffer. Therefore the Rice coder in this implementation aborts if the output becomes

larger than the input by simply making a copy of the input buffer to the output buffer, with a leading zero byte (making the output at most one byte larger than the input).

2.4 Lempel-Ziv (LZ77)

There are many different variants of the Lempel-Ziv compression scheme. The Basic Compression Library has a fairly straight forward implementation of the LZ77 algorithm (Lempel-Ziv, 1977) that performs very well, while the source code should be quite easy to follow.

The LZ coder can be used for general purpose compression, and performs exceptionally well for compressing text. It can also be used in combination with the provided RLE and Huffman coders (in the order: RLE, LZ, Huffman) to gain some extra compression in most situations.

2.4.1 Principle

The idea behind the Lempel-Ziv compression algorithm is to take the RLE algorithm a few steps further by replacing sequences of bytes with references to previous occurrences of the same sequences.

For simplicity, the algorithm can be thought of in terms of string matching. For instance, in written text certain strings tend to occur quite often, and can be represented by pointers to earlier occurrences of the string in the text. The idea is, of course, that pointers or references to strings are shorter than the strings themselves.

For instance, in the previous paragraph the string “string” is quite common, and replacing all occurrences but the first of that string with references would gain several bytes of saved storage.

A string reference is typically represented by:

- A unique marker.
- An offset count.
- A string length.

Depending on the coding scheme a reference can either have a fixed length or a variable length. The latter is often preferred since that allows the coder to trade reference size for string size (i.e. it may be worth the increased size in the reference representation if the string is long enough).

2.4.2 Implementation

One of the problems with LZ77 is that the algorithm requires exhaustive string matching. For every single byte in the input data stream, every previous byte in the stream has to be considered as a possible starting point for a matching string, which means that the compressor is very slow.

Another problem is that it is not very easy to tune the representation of a string reference for optimal compression. For instance, one has to decide if all references and all non-compressed bytes should occur on byte boundaries in the compressed stream or not.

The Basic Compression Library uses a very straight forward implementation that guarantees that all symbols and references are byte aligned, thus sacrificing compression ratio, and the string matching routines are not very optimized (there are no caches, history buffers or other similar tricks to gain speed), which means that the routines are *very* slow.

On the other hand, the decompression routines are very simple and fast.

An attempt to speed up the LZ77 coder has been made, which uses an index array that speeds up the string matching process by a fair amount. Still, it is much slower than any conventional compression program or library.¹

¹On a 2 GHz CPU, the compression speed is usually in the order of 40 KB/s

Chapter 3

Compiling

There is a Makefile included for the GNU C compiler (`gcc`). Just run `make` from the `src` directory, and you will get a file called `libbcl.a`, which is a static link library that you can copy to your compiler's `lib` directory. The library has been compiled with MinGW 2.0 under Windows 2000 without any problems, and it should compile under any environment with GCC support out of the box (e.g. Linux, Mac OS X, DOS/DJGPP, etc).

To compile the Basic Compression Library with an alternate compiler, you can either change the Makefile as appropriate, or simply add the `.c/.h` files to your own project.

Chapter 4

Library API Reference

All functions act on input and output buffers, which contain any kind of binary data. All sizes are given in number of bytes. The output buffer usually has to be slightly larger than the input buffer, in order to accommodate potential overhead if the input data is difficult to compress.

4.1 RLE_Compress

Syntax:

```
outsize = RLE_Compress(in,out,insize)
```

outsize Size of output buffer after compression
in Pointer to the input buffer (uncompressed data)
out Pointer to the output buffer (compressed data)
insize Size of input buffer

The output buffer must be able to hold $insize \times \frac{257}{256} + 1$ bytes.

4.2 RLE_Uncompress

Syntax:

```
RLE_Uncompress(in,out,insize)
```

in Pointer to the input buffer (compressed data)
out Pointer to the output buffer (uncompressed data)
insize Size of input buffer

The output buffer must be able to hold the entire uncompressed data stream.

4.3 Huffman_Compress

Syntax:

```
outsize = Huffman_Compress(in,out,insize)
```

outside Size of output buffer after compression
in Pointer to the input buffer (uncompressed data)
out Pointer to the output buffer (compressed data)
insize Size of input buffer

The output buffer must be able to hold $insize \times \frac{101}{100} + 384$ bytes.

4.4 Huffman_Uncompress

Syntax:

```
Huffman_Uncompress(in, out, insize, outside)
```

in Pointer to the input buffer (compressed data)
out Pointer to the output buffer (uncompressed data)
insize Size of input buffer
outside Size of output buffer

The output buffer must be able to hold *outside* bytes.

4.5 Rice_Compress

Syntax:

```
outside = Rice_Compress(in, out, insize, format)
```

outside Size of output buffer after compression (in bytes)
in Pointer to the input buffer (uncompressed data)
out Pointer to the output buffer (compressed data)
insize Size of input buffer (in bytes)
format Word format (see rice.h)

The output buffer must be able to hold $insize + 1$ bytes.

4.6 Rice_Uncompress

Syntax:

```
Rice_Uncompress(in, out, insize, outside, format)
```

in Pointer to the input buffer (compressed data)
out Pointer to the output buffer (uncompressed data)
insize Size of input buffer (in bytes)
outside Size of output buffer (in bytes)
format Word format (see rice.h)

The output buffer must be able to hold *outside* bytes.

4.7 LZ_Compress

Syntax:

```
outsize = LZ_Compress(in,out,insize)
```

outsize Size of output buffer after compression
in Pointer to the input buffer (uncompressed data)
out Pointer to the output buffer (compressed data)
insize Size of input buffer

The output buffer must be able to hold $insize \times \frac{257}{256} + 1$ bytes.

4.8 LZ_CompressFast

Syntax:

```
outsize = LZ_Compress(in,out,insize,work)
```

outsize Size of output buffer after compression
in Pointer to the input buffer (uncompressed data)
out Pointer to the output buffer (compressed data)
insize Size of input buffer
work Pointer to a temporary buffer (internal working buffer)

The output buffer must be able to hold $insize \times \frac{257}{256} + 1$ bytes, and the work buffer must be able to hold $insize + 65536$ unsigned integers.

4.9 LZ_Uncompress

Syntax:

```
LZ_Uncompress(in,out,insize)
```

in Pointer to the input buffer (compressed data)
out Pointer to the output buffer (uncompressed data)
insize Size of input buffer

The output buffer must be able to hold the entire uncompressed data stream.

Chapter 5

License

Copyright © 2003-2004 Marcus Geelnard

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.