General Remarks on GZIP Compression

Faculty of Physics
University of Regensburg

Thomas M. Karl

2021

I. GZIP Compression

In this chapter, we describe the *Gzip* data compression in detail. In particular, the file format and the *Deflate Algorithm* are explained. To illustrate the necessary fundamentals of data compression, we start by introducing information entropy and entropy coding on a theoretical level. In the following part we discuss the details of the LZ77 and *Huffman* encoding algorithms, which are the essential building blocks of the *Deflate Algorithm*. We describe the *Deflate Algorithm* itself and its decompression counterpart, the *Inflate Algorithm*. The chapter is concluded with technical details of the *Gzip* file format.

I 1. Information Entropy and Huffman Trees

I 1.1. Entropy Coding

Entropy coding is an example of lossless data compression. Each character in a given text is mapped to a sequence of bits. The minimum number of bits that is needed to distinguish between the characters is determined by the entropy of the text.

Claude Shannon defines the entropy [9] H of a discrete random variable X over a finite set of characters $Z = \{z_1, z_2, \ldots, z_m\}$. Let $I(z) = -\log_2 p_z$ be the amount of information one can get of an event that occurs with probability p. The probability that a character $z \in Z$ occurs is denoted by $p_z = P(X = z)$. The entropy of a character is the expected value of its information,

$$H_1 = E[I] = \sum_{z \in Z} p_z I(z) = -\sum_{z \in Z} p_z \log_2 p_z = -\sum_{i=1}^{m} p_i \log_2 p_i \,, \tag{I.1}$$

with $p_i = p_{z_i}$.

The entropy for an arbitrary word w consisting of n characters is defined as

$$H_n = -\sum_{w \in Z^n} p_w \log_2 p_w \,, \tag{I.2}$$

where $p_w = P(X = w)$ is the probability of an occurrence of a word w. As $n \to \infty$ the overall entropy reads,

$$H = \lim_{n \to \infty} \frac{H_n}{n} \,. \tag{I.3}$$

In order to calculate a normalized measure for a discrete distribution, the maximum entropy of the system can be used assuming that p_i are equally distributed. Let $N = |Z|$ be the number of characters in a source X over the alphabet Z. If

$$p_i = \frac{1}{N} \quad \forall \, p_i \,,$$

the maximum entropy reads

$$H_{\max} = -\sum_{i=1}^{N} \frac{1}{N} \log_2 \frac{1}{N} = \log_2 N \,. \tag{I.4}$$

Normalizing the entropy with the maximum value of the same system yields

$$\frac{H}{H_{\max}} = -\sum_{i=1}^{|Z|} p_i \cdot \frac{\log_2 p_i}{\log_2 N} = -\sum_{i=1}^{|Z|} p_i \cdot \log_N p_i \leq 1 \,. \tag{I.5}$$

The following example explains entropy more intuitively. The roman alphabet consists of $N = 26$ characters. The maximum entropy is $H_{\max} = \log_2 26 \approx 4.7$ per character. Since we chose 2 as basis of the logarithm the unit of the entropy is a bit. Replacing the probabilities with the actual relative frequencies of the letters in a standard English text (excluding blanks and other characters) yields a slightly smaller entropy $H \approx 4.06$[1]. The overall redundancy

[1] For more information see F. G. Guerrero [4].

divided by the entropy $N \cdot (H_{\max} - H)/H \approx 4.08$ gives the number of characters that yield such redundancy. Hence, four letters of the alphabet are actually unnecessary.

Naturally, we cannot simply delete four characters and expect that the meaning of a text does not change. Instead we could define a new alphabet with only 22 characters in such a way that the original alphabet can be replaced without information loss.

Entropy coding is a generic term for representations with the intention of reducing information entropy. An algorithm which tries to find an encoding that reduces the entropy is by definition a lossless compression algorithm[2]. In the given example we could reduce the size of an English text to 14% without information loss.

A prominent way and historically one of the first to compute an entropy coding is with the help of *Huffman* trees.

I 1.2. Huffman Coding

Huffman Coding is a form of entropy coding developed by David Huffman in 1952 [6]. It assigns codewords with variable length to a fixed number of characters in a source. The idea is to represent characters that occur more frequently with codewords of smaller bit length. In order to decode the codewords unambiguously they have to fulfill the Kraft–McMillan inequality and have to be free of prefixes. The first condition is a necessary and sufficient condition for the existence of a prefix code [1]. Prefix code or free of prefixes means that no codeword can be the beginning of another one.

The algorithm uses a k-nary tree as representation of the code. A k-nary tree is a tree with exactly k children per node. The leaf nodes represent the characters while the path from root to leaf represents the codeword. The tree is constructed from leaf to root (bottom-up). The unique mapping from character to codeword and vice versa is called codebook.

The following must be known *a priori*:

- Z: the set of characters that are to be encoded

[2]Such an algorithm does not necessarily minimize the entropy.

- p_z: probability with which a character $z \in Z$ occurs
- C: the set of characters that form the codewords
- $N = |C|$: cardinality of C

The tree is constructed as follows. Construct a node for each character by assigning the relative frequency (the probability) of that character. Repeat the following procedure to construct a full tree: Choose N subtrees with the smallest probabilities that was assigned to their root nodes. Prioritize the subtree that is less deep. Combine the two subtrees to a single tree by connecting them to a new, additional node. Assign the summed probability of the two subtrees to the new node.

In the second step, codewords are assigned to the original characters unambiguously. Start by assigning a code character in C to every connection from a specific node to its child nodes. Then the codewords for a specific character, which is represented by a leaf node, is given as follows. Beginning at the root, traverse the tree down to the character. The codeword consists of the visited codesigns in that order.

The encoding itself is rather simple. Read a character and retrieve the codeword from the codebook. Simply replace each character with the corresponding codeword.

In the following, we present an example for the application of *Huffman* encoding.

The following text shall be encoded:

$$\text{a a c a b a b b c d}\,.$$

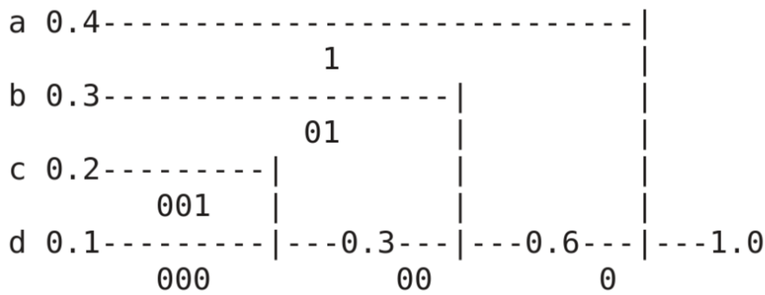
A straight-forward way to represent four different characters with $\log_2 4 = 2$ bits per symbol (the maximum entropy per character) is by numerating them from 0 to 3 with binary numbers. A naive encoding would yield

a	a	c	a	b	a	b	b	c	d
00	00	10	00	01	00	01	01	10	11 .

The set of characters is $Z = \{a, b, c, d\}$. We choose the binary code as codealphabet, *i. e.* the set of codesigns is $C = \{0, 1\}$ and $N = |C| = 2$. The relative frequencies of the characters are

$p_a = 0.4$, $p_b = 0.3$, $p_c = 0.2$ and $p_d = 0.1$.

The construction of the *Huffman* tree,



yields the codebook,

a: 1

b: 01

c: 001

d: 000.

The original text can be encoded as

a	a	c	a	b	a	b	b	c	d
1	1	001	1	01	1	01	01	001	000 .

Since the calculation of the relative frequencies can be very time consuming, an estimated tree can be provided in advance. Such a so-called static *Huffman* tree can be set up empirically *e. g.* by analyzing a specific file type and assuming that these relative frequencies are sufficient estimates for all files of that kind.

The average codeword length is

$$\bar{l} = \sum_{x \in X} p_x l_x \, . \tag{I.6}$$

Additionally, $\bar{l}$ has a lower and an upper bound [10]

$$H(X) \leq \bar{l} \leq H(X) + 1\,. \tag{I.7}$$

This implies that a codeword is on average represented at least with the same number of digits than its amount of information, but at most with only one more.

The *Huffman* representation in the example encodes each character with

$$\begin{aligned}\bar{l} &= 0.4 \cdot 1 + 0.3 \cdot 2 + 0.2 \cdot 3 + 0.1 \cdot 3 \\ &= 0.4 + 0.6 + 0.6 + 0.3 = 1.9\end{aligned}$$

bits. The entropy is

$$\begin{aligned}H(X) &= -(0.4 \cdot \log_2 0.4 + 0.3 \cdot \log_2 0.3 + 0.2 \cdot \log_2 0.2 + 0.1 \cdot \log_2 0.1) \\ &= 0.529 + 0.521 + 0.464 + 0.332 = 1.85\end{aligned}$$

bits per character. The *Huffman* coding reduces the length of the input to $\frac{1.9}{2} = 95\%$, whereas the maximum that can be achieved with entropy coding would be $\frac{1.85}{2} = 92.5\%$.

Huffman Coding gives in general high compression ratios when the probabilities are unequally distributed. This is usually true for source code, where special characters like {, ; or # are much more likely to occur than *e. g.* spaces. However, when the distribution is close to a uniform one, almost no compression can be achieved, since the entropy is close to the maximum.

Decoding

In order to decode characters the *Huffman* tree has to be traversed simply in the opposite direction. With each incoming bit the tree has to be followed beginning at the root node until a leaf node is reached. The codeword can be replaced with the original character. Along with the encoded output the *Huffman* tree has to be stored additionally.

I 2. Redundancy Elimination with LZ77

The main disadvantage of entropy coding is the lack of redundancy elimination. Redundancy is the occurrence of a sequence in a text more than once. Since X was defined (in I 1.1) as a discrete random variable (implying that it has no memory of previous characters), it is impossible to take such repetitions into account. For example, in an English text one can easily replace frequent words with a special character. With that, probably, a better compression than with minimum entropy coding will be achievable. Based on this principle, Abraham Lempel an Jacob Ziv developed an algorithm for lossless data compression in 1977 (LZ77) [12]. It is historically the first method that makes use of repetition of sequences and can be applied to any data.

Mathematically speaking, a LZ77 factorization x is a decomposition of a sequence of characters into non-empty sequences $w_1, w_2, \ldots, w_k$ that obey the rules:

- $x = w_1 w_2 \cdots w_k$,
- for each w_j with $1 \leq j \leq k$,
  - w_j is a new character α, which is not in $w_1 \cdots w_{j-1}$ or
  - w_j is the longest sequence that is in $w_1 \cdots w_j$ at least two times.

The algorithm generates a sequence of triples from which the original text can be reproduced. A triple for a specific factor w_j is of the form of (**pos**,**len**,λ), where

- **pos** is the position of the previous occurrence of w_j in x (or 0 if none exists),
- **len** is its length (or 0, if w_j is a new character),
- and λ is the first character to be considered a mismatch.

The triples are also called symbols. The position is always relative to the right or left border of the buffer, which is defined by the implementation. This has also to be considered when the output is decoded.

The length of the book is usually not allowed to exceed a specified size due to technical rea-

sons. A longer codebook also yields longer times for searching prefixes. Therefore, common implementations often use a sliding window, which restricts the codebook and the length of the examined input sequence (preview buffer). In each step of the algorithm an input stream is shifted into the preview buffer, where the length of the shift is the length of the matching sequence plus an additional one. Therefore, redundant triples are avoided. Otherwise, a new character would always be added as a single one.

The following example demonstrates the application of the LZ77 algorithm to compress the sequence **aacaacabcabaaac**. An additional end-of-file (**eof**) character is attached in order to signalize the algorithm to stop.

Table I.1 shows the procedure using a preview buffer of length 10 and a codebook of length 12. Each line corresponds to one step of the algorithm. The right column is filled with the output symbols after the step is completed.

12	11	10	9	8	7	6	5	4	3	2	1	0	1	2	3	4	5	6	7	8	9	output
												a	a	c	a	a	c	a	b	c	a	(0,0,a)
											a	a	c	a	a	c	a	b	c	a	b	(1,1,c)
									a	a	c	a	a	c	a	b	c	a	b	a	a	(3,4,b)
				a	a	c	a	a	c	a	b	c	a	b	a	a	a	c		eof		(3,3,a)
a	a	c	a	a	c	a	b	c	a	b	a	a	a	c				eof				(12,3,eof)

Table I.1.: Visualization of LZ77 compressing the sequence **aacaacabcabaaac eof**. The codebook is on the left side, the preview buffer on the right. Matching sequences found in the codebook are colored in green, the corresponding sequences in the buffer red. An overlap of both is marked yellow. The character that terminates the matching sequence is marked purple. [8]

1. The first character is always unknown. The output is $(0, 0, a)$. The sequence is shifted by one into the codebook.

2. The sequence a (red) in the buffer matches a sequence in the codebook at position 1 (green). The character ending the matching sequence (the mismatch character) is c (purple). The length of the matching sequence is 1. Therefore, the output is $(1, 1, c)$ and the input is shifted until c is in the codebook.

3. The sequence $aaca$ can be found in the codebook. The matching sequence at position 3 and the sequence to be shifted are overlapping (yellow). The length of the matching sequence is 4, the first mismatch is b. Therefore, the output is $(3, 4, b)$.

4. The second a terminates the matching sequence. The output is $(3, 3, a)$.

5. The last step is straight forward. The **eof** character is treated exactly as any other character. In addition, it signalizes the algorithm to abort compressing.

The final encoded output is $(0, 0, a)(1, 1, c)(3, 4, b)(3, 3, a)(12, 3, \text{eof})$. The entire codebook can be searched for the longest match in order to guarantee the best compression. Some implementations of the LZ77 algorithm like *Gzip* provide options to terminate the steps sooner (search intensity) in order to decrease computation time, *e. g.* if a match of a specified minimum length is found.

The computational complexity is $\Theta(N)$, where N is the length of the input sequence, since the sizes of the preview buffer and the codebook are fixed. Searching matches in a very short sequence does not yield a high contribution for the overall computation time.

Decompression

The original text can be reconstructed only from the triples without usage of a codebook. All triples are unique. The search intensity of the decompression does not affect the decompression. The complexity is linear, since the number of reconstructions is in the worst case exactly the length of the original text ($\Theta(N)$). Implementing the decompression is much simpler and shall be illustrated on the basis of the previously computed symbols (tab I.2).

input	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
(0,0,a)																a
(1,1,c)														a	a	c
(3,4,b)									a	a	c	a	a	c	a	b
(3,3,a)					a	a	c	a	a	c	a	b	c	a	b	a
(12,3,eof)	a	a	c	a	a	c	a	b	c	a	b	a	a	a	c	eof

Table I.2.: Visualization of LZ77 decompressing the sequence $(0, 0, a)(1, 1, c)(3, 4, b)(3, 3, a)(12, 3, \text{eof})$. There are no buffers necessary. The same color coding as in table I.1 applies. [8]

1. **(0,0,a)** The first entry is always trivial. It is simply replaced with the first character.

2. **(1,1,c)** Beginning at position 1 in the sequence decoded in the previous step, the first character is duplicated and a c is attached.

3. **(3,4,b)** Beginning at position 3 a sequence of four consecutive characters is duplicated. Since only three exist, the fourth character to be duplicated (yellow) is again the character at position 3. Thereafter, the *b* is attached and the decoded sequence reads **aacab**.

4. **(3,3,a)** Three letters at position 3 are duplicated and attached with an additional *a*. The sequence reads **caba**.

5. **(12,3,eof)** Three letters at position 12 in the previous step are duplicated. The **eof** character signalizes the algorithm to abort decompressing and is ignored. The sequence reads **aac**.

The entire output after decompression reads **aacaacabcabaaac**, which was exactly the input sequence for the compression algorithm.

I 3. Deflate Algorithm

The basic idea behind the *Deflate Algorithm* is the combination of two different compression methods. In the first step, the algorithm reduces redundancy with LZ77. In the second step, *Huffman Coding* is applied in order to obtain an efficient binary representation. The algorithm is specified in the documentation [2].

There are three different parameters that must be declared *a priori*. These properties have a great impact on the compression ratio.

- Size of the LZ77 codebook: The locating of redundant sequences is more promising for larger codebooks. However, the computation time also increases. Prominent implementations like *Gzip* use 32 KiB.
- Search intensity: The algorithm can either use the first matching sequence of a specific length or keep going on in order to find longer ones.
- *Huffman* trees: A static tree can be declared in the beginning. A tree can also be created dynamically out of the given data. The latter case will probably end up with higher compression ratios, but induces additional computation time.

Redundancy Reduction

In the specification of the *Deflate Algorithm* the LZ77 algorithm is not explained in detail. The document only refers to the IEEE publication of Lempel an Ziv [12] and suggests an implementation that is not patented. Therefore, many variations of LZ77 are also in use for implementations of the *Deflate Algorithm*.

The LZ77 compression is not applied to the entire text, but only to a single block of data in order to save computation time. The method is often implemented by inserting a special end-of-block character (**eob**), which signalizes the LZ77 compression to stop. After one block is compressed the LZ77 algorithm is repeated with the next block. It is highly recommended to follow some additional rules.

- The implemented compression algorithm (compressor) terminates a block when it deter-

mines that starting a new block with fresh trees would be useful, or when the block size fills up the memory buffer.

- The compressor uses a chained hash table to find duplicated strings.
- The compressor searches the hash chains starting with the most recent strings to favor small distances and thus take advantage of the *Huffman* encoding.

An implementation is not forced to obey these rules in order be compliant with the *Deflate* specification. One match of an arbitrary sequence consists of a length (ranging from 3 to 258 bytes) and a distance (1 to 32 768 bytes). Such a reference can extend to multiple blocks, but has to reside within a distance of the size of the codebook in the decompressed data.

Entropy Coding

In the second stage, an entropy coding according to *Huffman* is applied on the outputted triples. The *Huffman* trees for each block are independent of those for previous or subsequent blocks. The *Huffman* trees themselves are compressed using *Huffman* encoding. Note that according to the specification the *Huffman* codes in the *Deflate* format for the various alphabets must not exceed certain maximum code lengths. There are two additional requirements for *Huffman* codes:

1. All codes of a given bit length have lexicographically consecutive values.
2. Shorter codes lexicographically precede longer codes.

The representation of the trees is probably the most complicated part of the specification. Each type of value (literals, distances, and lengths) in the compressed data is represented using a *Huffman* code. One code tree for literals and lengths and a separate code tree for distances are used. Literals and lengths are merged into a single alphabet. The literal/length tree stores 288 different symbols:

- 0 to 255: character from 0 to 255 in ASCII code,
- 256: end-of-block symbol (**eob**),

- 257 to 285: length of 3 to 258 bytes in conjunction with extra bits (example: code 269 can represent with two extra bits $1 + 0b11 = 4$ different lengths from 19 to 22),
- 286 to 287: reserved, not in use despite being part of the tree.

The distance tree is defined in a similar way. Depending on the code, it is possible that some extra bits are read in order to calculate the final distance. The tree consists of 32 different symbols:

- 0 to 3: distance 1 to 4,
- 4 to 5: distance 5 to 8, 1 bit extra,
- 6 to 7: distance 9 to 16, 2 bits extra,

  $\vdots$,
- 28 to 29: distance 16.385 to 32.768, 13 bits extra,
- 30 to 31: reserved, not in use despite being part of the tree.

For symbols from 2 to 29 the number of extra bits can be calculated as $\left\lfloor \frac{n}{2} - 1 \right\rfloor$.

This representation together with the extra bits is strictly specified [2].

Huffman trees can be defined globally before compression even starts (static) or individually for each block (dynamic). The difference is described in the following.

static Huffman codes

The *Huffman* codes for the two alphabets are specified and not stored explicitly in the output data. The *Huffman* code lengths for the literal/length alphabet are shown in table I.3 .

Lit Value	Bits	Codes		
0 - 143	8	00110000	through	10111111
144 - 255	9	110010000	through	111111111
256 - 279	7	0000000	through	0010111
280 - 287	8	11000000	through	11000111

Table I.3.: Code lengths for the literal/length alphabet

dynamic Huffman codes

The literal/length code appears before the distance code. For even more compactness, the code length sequences themselves are compressed using a *Huffman* code. The alphabet for code lengths is as follows:

- 0 - 15: Represent code lengths of 0 - 15,
- 16: Copy the previous code length 3 - 6 times. The next 2 bits indicate repeat length (0 = 3, ... , 3 = 6),
- 17: Repeat a code length of 0 for 3 - 10 times (3 bits of length),
- 18: Repeat a code length of 0 for 11 - 138 times (7 bits of length).

A code length of zero indicates that the corresponding symbol in the literal/length or distance alphabet will not occur in the block and should not participate in the *Huffman* code construction algorithm. If only one distance code is used, it is encoded using one bit, not zero bits. In this case, there is a single code length of one with one unused code. One distance code of zero bits means that there are no distance codes used at all (the data is all literals). [2]

Format

Finally, we can discuss the format of the *Deflate* stream. A complete stream consists of a series of compressed or uncompressed blocks. The first 3 bits of each block have the following meaning. The first bit is a boolean number that states if the following block is the last one. The next two bits denote the method of coding:

- 00: uncompressed block, size between 0 and 64 KiBytes ,
- 01: compressed block, encoded with a predefined (static) *Huffman* tree,
- 10: compressed block, encoded with its own (dynamic) *Huffman* tree,
- 11: reserved, an occurrence in compressed data is treated as error.

The implementation may decide on its own which of the three methods should be used for the given input.

If the input was compressed, the utilized *Huffman* trees have to be provided. The static trees are usually hard-coded by the (de)compressor. Dynamic trees appear in a compact form immediately before the compressed data for that block and directly after the header in the following format:

- 5 Bits: HLIT, number of literal/length codes - 257 (257 - 286)
- 5 Bits: HDIST, number of distance codes - 1 (1 - 32)
- 4 Bits: HCLEN, number of code length codes - 4 (4 - 19)
- (HCLEN + 4) x 3 bits: code lengths for the code length alphabet in the order: 16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15

  These code lengths are interpreted as 3-bit integers (0-7). A code length of zero means the corresponding symbol (literal/length or distance code length) is not used.
- HLIT + 257 bits code lengths for the literal/length alphabet, encoded using the code length *Huffman* code
- HDIST + 1 bits code lengths for the distance alphabet, encoded using the code length *Huffman* code

All code lengths form a single sequence of HLIT + HDIST + 258 values.

In both static and dynamic cases the actual compressed data is stored with an additional end-of-block character (symbol 256).

It was already mentioned that it is up to the compressor when to terminate the compression procedure, *i. e.* the maximum size of the output data is not specified. The size for an uncompressed block is not allowed to exceed 64 KiBytes.

The length of an uncompressed block is stored in the next two bytes after the 3-bit header and its one's complement in the consecutive two bytes.

When one block is completed, the next block is compressed until the end of the input is reached.

I 4. Inflate Algorithm

In this section, we describe the decompression method which corresponds to the *Deflate Algorithm*. It does not require knowledge about block sizes or numbers, codebook lengths or search methods. An implementation must only provide the static *Huffman* tree used for compression. A pseudo code is already suggested in the specification [2].

```
do
    read block header from input stream.

    if stored with no compression
        skip any remaining bits in current partially
            processed byte
        read LEN, NLEN
        copy LEN bytes of data to output

    otherwise
        if compressed with dynamic Huffman codes
            read representation of code trees

        loop (until eob code recognized)
            decode literal/length value from input stream

            if value < 256
                copy value (literal byte) to output stream

            otherwise
                if value = eob (256)
                    break
                otherwise (value = 257..285)
```

```
24                          decode distance from input stream
                            move backwards distance bytes in the output stream
26                          copy length bytes from that position to the
                                output stream.
28              end loop
        while not last block
30
```

Listing I.1: Inflate Algorithm in pseudocode

I 5. GZIP File Format

The *Deflate Algorithm* has many parameters the compressor has to define. The most prominent tool that implements the algorithm is the *Gzip Compression Utility*, which is installed by default on most Linux distributions [5]. It was developed by Jean-loup Gailly and Mark Adler to replace the Unix *compress* utility [11]. A specific *Deflate* method is defined in *Gzip*, which writes compressed data to specified files (.gz) [3]. Figure I.1 shows the flowchart of the entire process. As in any compressed file, a header is required in order to store some meta information, as, for example, original file name or time stamp. This data is not relevant for the decompression and is not discussed further here. For more information see appendix **??**. Immediately after the header the entire *Deflate* stream is stored. The next four bytes are supposed to store a cyclic redundancy checksum of the original data according to ITU [7]. After decompression, the checksum can be calculated again and compared with the provided one in order to check the integrity of the data. The last 4 bytes store the length of the original file.

With four bytes the maximal number for the length is 2^{32} bytes $= 4\,$GiBytes. However, compressing larger files is possible. As a consequence this will result in a wrong file size and in a negative compression ratio. Decompression is still possible, since the decompressor does not need knowledge about the original file and simply aborts when all data is processed. Once the file is compressed, the original length cannot be deduced from the *Gzip* file.

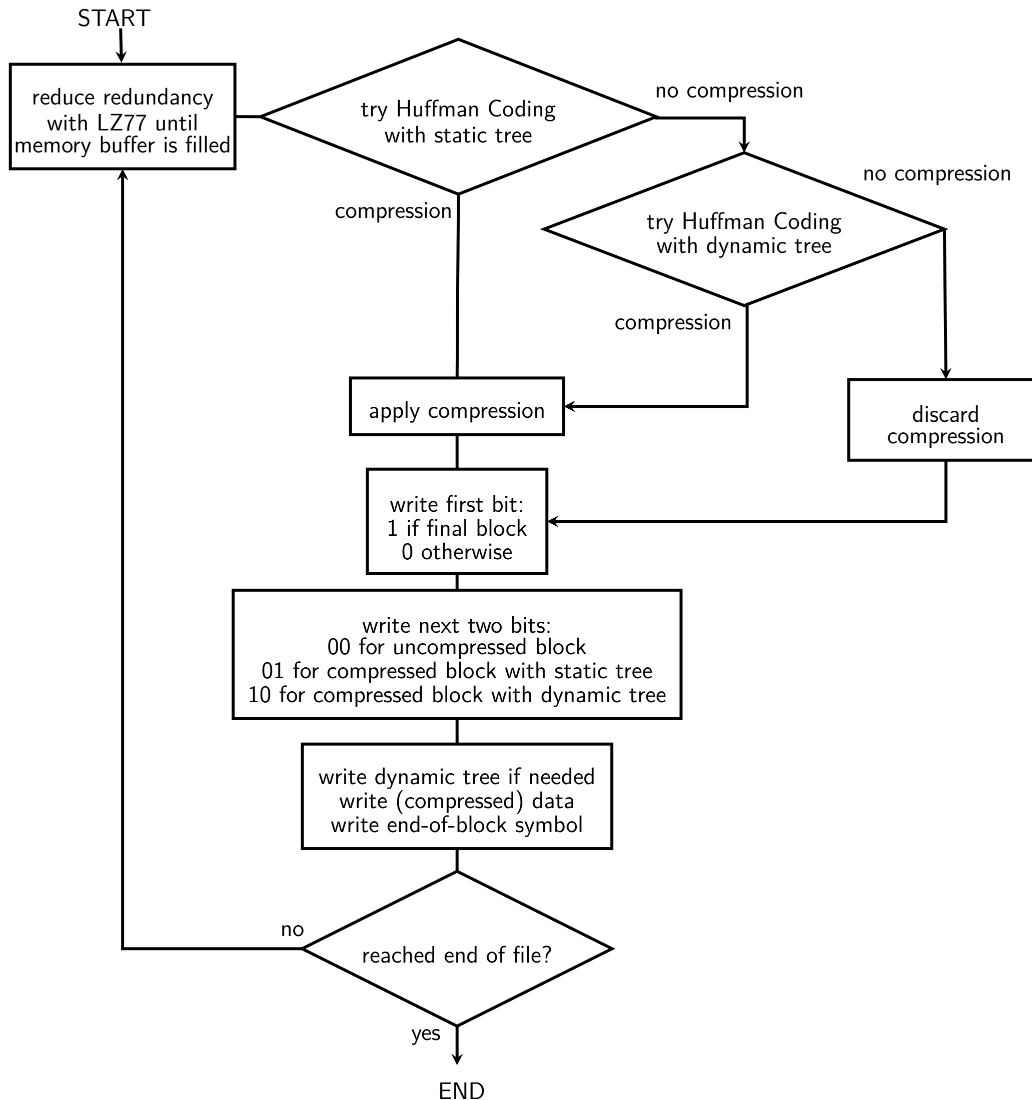


Figure I.1.: Flowchart of *Deflate* compression according to *Gzip*. The program reads the input data and tries to reduce redundancy with LZ77 until the memory buffer provided by the implementation is filled. The resulting lengths/symbols and distances are encoded using static *Huffman* trees. If the resulting compression did not reduce the size, a dynamic *Huffman* tree is applied. If there was still no reduction, the implementation discards the result and stores the block uncompressed. If the block was the last one, a 1 is written to the first bit of the block, else 0. The next two bit are either 00, 01 or 10, depending on the type of compression. The output data is written to disc including the dynamic tree if needed. If the first bit was actually 0, the algorithm starts all over again with the next block until the end of the input is reached.

Bibliography

[1] "Data Compression." In: *Elements of Information Theory*. John Wiley and Sons, Ltd, 2005. Chap. 5, pp. 103–158. ISBN: 9780471748823. DOI: https://doi.org/10.1002/047174882X.ch5. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/047174882X.ch5. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/047174882X.ch5.

[2] L. Peter Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*. RFC 1951. RFC Editor, May 1996, pp. 1–15. URL: https://tools.ietf.org/html/rfc1951.

[3] L. Peter Deutsch. *GZIP file format specification version 4.3*. RFC 1952. RFC Editor, May 1996, pp. 1–12. URL: https://tools.ietf.org/html/rfc1952.

[4] Fabio G. Guerrero. "A New Look at the Classical Entropy of Written English." In: *CoRR* abs/0911.2284 (2009). arXiv: 0911.2284. URL: http://arxiv.org/abs/0911.2284.

[5] *Gzip - GNU Project - Free Software Foundation*. Accessed: 2021-01-21. URL: https://www.gnu.org/software/gzip/.

[6] D. A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes." In: *Proceedings of the IRE* 40.9 (Sept. 1952), pp. 1098–1101. ISSN: 2162-6634. DOI: 10.1109/JRPROC.1952.273898.

[7] Telecommunication Standardization Sector of ITU. *ITU-T V.42 – Error-correcting procedures for DCEs using asynchronous-to-synchronous conversion*. Tech. rep. International Telecommunication Union, Mar. 2002. URL: https://www.itu.int/ITU-T/recommendations/rec.aspx?rec=5692&lang=en.

[8] *LZ77 - Wikipedia*. Accessed: 2021-01-26. URL: https://de.wikipedia.org/wiki/LZ77.

[9] C. E. Shannon. "A mathematical theory of communication." In: *The Bell System Technical Journal* 27.3 (July 1948), pp. 379–423. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[10] Tilo Strutz. *Bilddatenkompression : Grundlagen, Codierung, Wavelets, JPEG, MPEG, H.264*. 3., aktualisierte und erweiterte Auflage. Wiesbaden: Vieweg, May 2005.

[11] *The Open Group Base Specifications Issue 7*. Accessed: 2021-01-21. 2018. URL: `https://pubs.opengroup.org/onlinepubs/9699919799/`.

[12] J. Ziv and A. Lempel. “A universal algorithm for sequential data compression.” In: *IEEE Transactions on Information Theory* 23.3 (May 1977), pp. 337–343. ISSN: 1557-9654. DOI: `10.1109/TIT.1977.1055714`.