VŠB – Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Brotli Compression Algorithm (Simplified Version)

2020 Daniel Chýlek

Foreword

You are reading a simplified version of my masters thesis.

This version focuses on explaining Brotli and exploring the official implementation, and omits details about a custom implementation and experimental modifications to the official compressor. You can find the full thesis at `https://github.com/chylex/Brotli-Builder`.

I would like to thank doc. Ing. Jan Platoš, Ph.D. for leading and supervising the project.

Contents

List of Figures

List of Tables

1 Introduction

Brotli is a general-purpose lossless compression algorithm developed by Google, Inc. It defines a bit-oriented format inspired by DEFLATE[1], which is in essence a combination of LZ77 and Huffman coding. Brotli aims to replace DEFLATE in HTTP compression by providing better compression ratios and more flexibility than current standards.

Section 2 describes the compression corpus used for testing and validation, and performs a comparison between Brotli, and both current and upcoming HTTP compression standards.

Section 3 explains important compression techniques, and details their use in the Brotli format. It also introduces Brotli-specific concepts and terminology.

Section 4 explores the official compressor implementation and advanced features of the format. The first part points out differences between quality levels. The second part describes and evaluates official implementations of individual features.

2 Setup & Organization

2.1 Test Data

Brotli was designed with multilingual text and web languages encoded with the UTF-8 standard in mind, but it is still able to reasonably compress text in other encoding systems, and certain kinds of binary data.

A mixed corpus was built for testing and analysis of both the custom implementation, and individual features of the official compressor.

The corpus includes commonly used lossless compression corpora, which have a variety of text and binary formats, as well as a collection of multilingual texts and website resources. Files gathered outside of existing corpora were selected to represent the kind of data Brotli is intended for, and thus should benefit from its features and heuristics.

The corpus includes 169 files totaling ≈ `263 MiB` (median ≈ `54.5 KiB`). All files were made available at `https://github.com/chylex/Brotli-Builder/blob/master/Paper/Corpus.7z`.

- The Canterbury Corpus (`2.7 MiB`)
- The Silesia Corpus (`202 MiB`)
- A selection of files from Snappy Test Data (`1.7 MiB`)
  - fireworks.jpeg, geo.protodata, html, html_x_4, kppkn.gtb, paper-100k.pdf, urls.10K
- The Bible in various languages (`37 MiB`)
  - Arabic, Chinese (Simplified & Traditional), English, Hindi, Russian, Spanish
- `HTML`, `CSS`, and `JS` files from several of the most popular websites (`20.4 MiB`)
  - Baidu, Facebook, Google, Instagram, VK, Wikipedia, Yandex, YouTube

Additional information about where and how the files were obtained is included in the full thesis.

2.1.1 Brotli vs. gzip vs. zstd

As Brotli targets HTTP compression, it makes sense to compare it to `gzip`, the currently most used HTTP compression method, and `zstd`, a new compression standard developed by Facebook.

This comparison only considers out-of-the-box results for each quality level. Although both Brotli and `zstd` include options for using large amounts of memory to improve compression, their use would not be reasonable for websites. Additionally, with support for custom dictionaries in `zstd` and upcoming support in Brotli, website owners could put additional effort into optimization.

While this is only a total size comparison, for HTTP compression it is also important to consider compression speeds at each quality level. The highest quality levels may not be suitable for on-the-fly compression, but could be used to pre-compress and serve static content.

Figure 1 shows the size comparison between Brotli, `gzip 1.10`, and `zstd 1.4.4`.

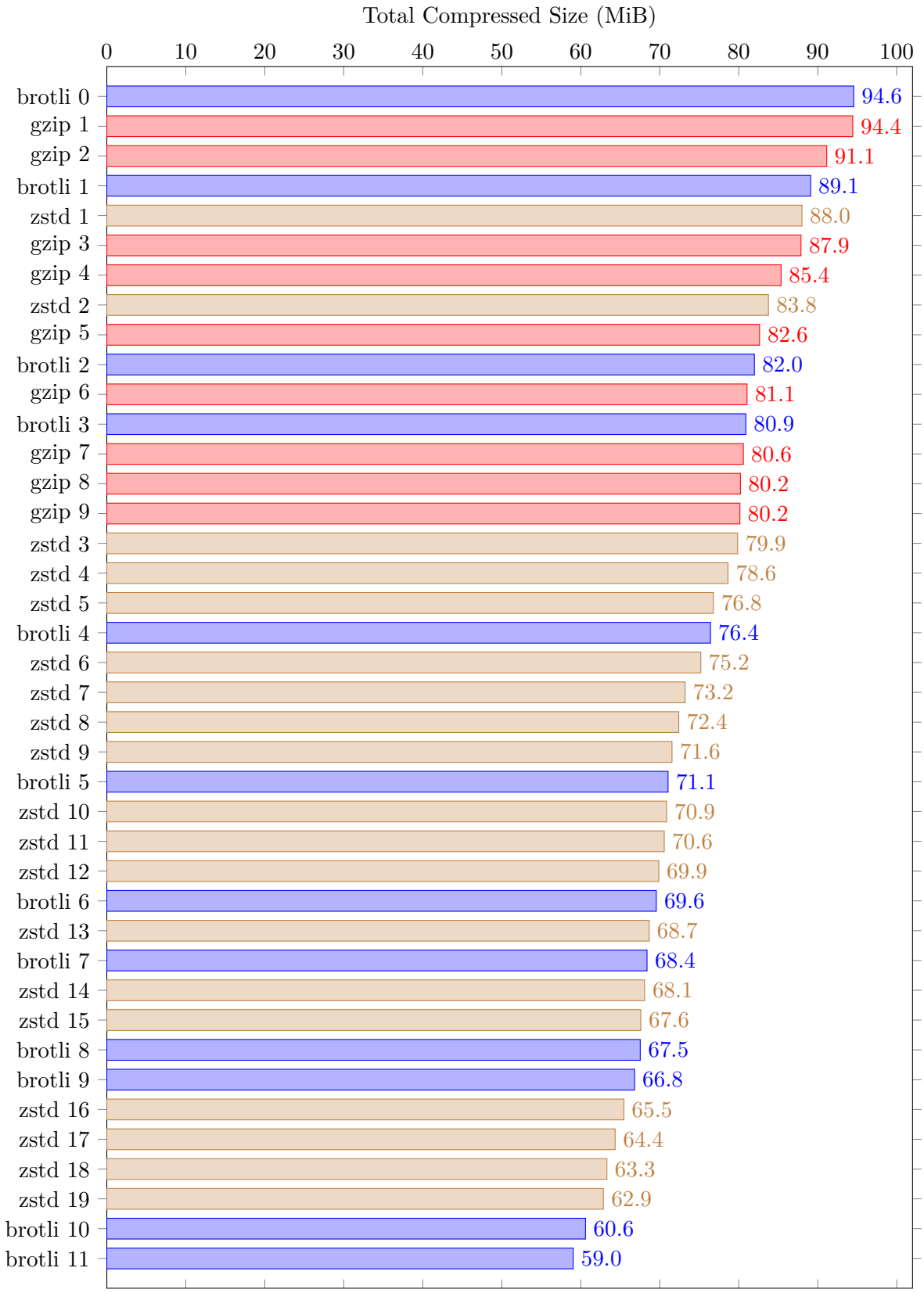


Figure 1: Test corpus size comparison between Brotli, gzip, and zstd.

3 Explaining Brotli

A Brotli compressed file or data stream comprises a *stream header* and a sequence of *meta-blocks*. A meta-block may be empty — optionally skipping part of the data stream — or it can contain output bytes in either uncompressed or compressed form. In the interest of brevity, assume that any mentions of meta-blocks, which do not explicitly specify their type, concern compressed meta-blocks.

Each meta-block begins with a header describing its type, size of uncompressed output (with an upper limit of `16 MiB`), and other type-dependent metadata. The header is immediately followed by a data section. In compressed meta-blocks, the data section is a sequence of commands, and the header includes all information needed to decode these commands.

To understand Brotli, we will look at the pieces that form a compressed meta-block, and how they fit together to generate uncompressed output bytes. The next few sections briefly introduce important concepts, many of which are also found in other kinds of compression algorithms and thus are not unique to Brotli.

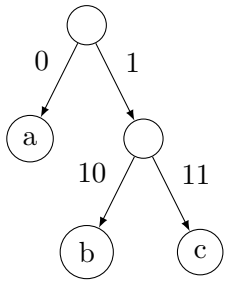
3.1 Huffman Coding

Binary Huffman coding encodes *symbols* of an *alphabet* using variable-length bit sequences[2]. The intention is that we can take a sequence of symbols, and construct a code that represents frequent symbols with short bit sequences, and infrequent symbols with long bit sequences. This by itself is a form of compression, and can be used to compress files by treating individual byte values as symbols.

Brotli makes heavy use of Huffman coding to encode not only individual bytes of the uncompressed output, but also various special codes that command the process of decompression.

We will define a Huffman tree as a full binary tree, where each leaf represents one symbol. Given a sequence of input bits, we start traversing the tree from its root, go down the left branch whenever we encounter a 0, go down the right branch whenever we encounter a 1, and output a symbol after reaching a leaf node. If any input bits remain, the traversal restarts at the root.

An example of such tree with a 3-symbol alphabet $\{a, b, c\}$ is shown in figure 2. In this example, the bit sequence 001011 would unambiguously map to the symbol sequence *aabc*.



a	0
b	10
c	11

Figure 2: Huffman tree containing paths to 3 symbols.

3.2 Intermediary Codes

Oftentimes, we want to encode a potentially very large range of values, with the assumption that small values are more likely to appear than large values.

Based on this assumption, Brotli defines several sets of codes used in various parts of the format. We will call them *intermediary codes*.

An intermediary code has a range of possible values. To obtain a concrete value within that range, we read a certain amount of *extra bits* from the bit stream, and add their value to the lower bound of the range. The difference between the upper bound and lower bound is therefore always a power of two.

Figure 3 demonstrates how an example set of 4 intermediary codes could be used to encode values between 1 and 21. Codes and their respective lower bounds are highlighted with different colors.

Code	Range	Difference	Extra Bits
00	1–1	$0 = 2^0$	0
01	2–3	$1 = 2^1$	1
10	4–12	$8 = 2^3$	3
11	13–21	$8 = 2^3$	3

Example	Value
00	1
01 0	$2 + 0 = 2$
01 1	$2 + 1 = 3$
10 000	$4 + 0 = 4$

Example	Value
10 001	$4 + 1 = 5$
10 111	$4 + 8 = 12$
11 000	$13 + 0 = 13$
11 111	$13 + 8 = 21$

Figure 3: Example of an intermediary code set on the left, and several bit sequences and their calculated values on the right.

3.3 Sliding Window

A sliding window is a fixed-size buffer containing the most recently generated uncompressed output, which can be referenced to repeat previously seen data. This technique is the backbone of LZ77[3], which has in turn influenced DEFLATE and eventually Brotli.

The Brotli format specification allows for windows sizes ranging from approximately `1 KiB` to `16 MiB`, although more recent versions of Brotli unofficially extend the size limit up to `1 GiB` which could be useful for large files outside HTTP compression.[1]

3.4 Static Dictionary

A dictionary holds sequences of bytes. In dictionary-based compression, sequences of bytes are replaced with references to matching sequences in the dictionary — for example, if the dictionary is a contiguous stream of bytes, a reference may be encoded as a $\langle\, position, length \,\rangle$ pair.

A sliding window can be considered a kind of *dynamic dictionary*, as its referenceable contents change with each newly output byte. In contrast, a *static dictionary* is immutable during the entire process of decompression.[4]

[1] `https://groups.google.com/forum/#!topic/brotli/aq9f-x_fSY4`

Although static dictionaries are inflexible and target only certain kinds of data — a dictionary of English words should work best when compressing an English book — if the same dictionary is used to compress multiple files, the overhead from storing it or obtaining it for the first time becomes negligible — for example, if a website uses the same dictionary for all its resource files, it only needs to be downloaded once.

Brotli defines a static dictionary with 13 504 *words*. A word is an arbitrary sequence of bytes, however most commonly they are words or phrases from spoken languages, and strings found in markup and programming languages used on the web.[2] Words are grouped by their length, which ranges from 4 to 24 bytes, and a dictionary reference is made of the word's length group and its index within that group.

This dictionary is part of the Brotli standard and can be used in all Brotli compressed files. Brotli intends to also support custom dictionaries as an extension of the Brotli format, however at the time of writing the new standard has not been finalized yet.

3.4.1 Word Transformations

One distinct feature of the static dictionary in Brotli is its word transformation system. A transform applies one of the available preset functions to the word itself, and may also add a prefix and/or a suffix. There are 121 different transforms applicable to all words, for a total of 1.6 million words.[3]

These are the available functions:

- **Identity** does not change the word
- **Omit First 1–9** removes the first 1–9 bytes of the word
- **Omit Last 1–9** removes the last 1–9 bytes of the word
- **Ferment All** splits the word into 1–3 byte UTF-8 code points, and performs a bitwise operation[4] on all code points
- **Ferment First** is similar, but it only performs the operation on the first code point

In a dictionary reference, the transform ID (0–120) is encoded as part of the index within its word length group. For the 4-byte word length group containing 1024 words, indices 0–1023 use transform ID 0, indices 1024–2047 use transform ID 1, and so on.

[2]The dictionary is part of Brotli's focus on HTTP compression. According to its developers, the words were selected from a multilingual web corpus of 93 languages[5].

[3]While 1 633 984 is the total amount of words representable by a dictionary reference, listing every possible word shows that only 1 399 565 are unique.

[4]The bitwise operation has no unifying meaning in terms of alphabetical characters, but we can still observe certain patterns that exploit how UTF-8 is laid out, whether by intention or coincidence. For all 1-byte code points, the transformation converts the lower case letters a–z to upper case. For some 2-byte code points, it toggles case of certain accented and non-latin alphabet letters. Note that the function does not handle 4-byte code points at all, and instead treats them as 3-byte code points — the 4-byte case would be triggered by 3 words in the dictionary, but those words are patterns of bytes (such as four `255` bytes followed by four `0` bytes) rather than words from languages.

To better understand the capability of the transformation system, table 1 shows a few concrete transforms defined in the format, each with an example word transformation:

Table 1: Concrete static dictionary transforms shown as examples of the transform system.

ID	Function	Prefix	Suffix	Example
0	Identity			`time` → `time`
1	Identity		␣	`time` → `time`␣
2	Identity	␣	␣	`time` → ␣`time`␣
3	Omit First 1			`time` → `ime`
9	Ferment First			`time` → `Time`
49	Omit Last 1		`ing`␣	`time` → `timing`␣
67	Identity	`.`	`(`	`time` → `.time(`
107	Ferment All		`.`	`time` → `TIME.`

You may notice in these examples that the transformations tend to be skewed towards sentence structures found in the English language, as well as constructs and special characters found in web languages, again alluding to the intended use in HTTP compression.

3.5 Insert & Copy Commands

Insert&copy commands are a fundamental part of a meta-block data section. Each insert&copy command generates uncompressed output in two parts.

The *insert* part 'inserts' a set amount of *literals* into the output. A literal is a byte — integer between 0 and 255. The amount of literals is determined by a parameter called *insert length*.

The *copy* part does one of two things, based on the *copy length* and *distance* parameters. Let $distance_{max}$ be the smaller of the two values *sliding window bytes* and *output bytes*, then:

- If $distance \leq distance_{max}$, the command references previously seen output, where *distance* is the position in the sliding window, and *copy length* is the amount of bytes to copy into the output. We will call references to previously seen output *backward references*.
- If $distance > distance_{max}$, the command references a dictionary word, where *copy length* is the word length group, and $(distance - distance_{max} - 1)$ determines the word index and transform ID. We will refer to them as *dictionary references*.

In the bit stream, each command begins with a *length code*, which encodes the *insert length* and *copy length* using two intermediary codes — *insert code* and *copy code*. The command continues with a sequence of literals, and usually ends with a *distance code* that determines the *distance*. Length codes, literals, and distance codes are encoded using their own separate Huffman trees. Sections 3.7 and **??** will talk about their use of Huffman trees in greater detail.

Because a meta-block header stores the amount of uncompressed bytes generated by its data section, if that amount is reached during the *insert* part of a command, the *copy* part is omitted.

As a side note, the *copy* part of an insert&copy command is allowed to reference output that will be generated by the command itself. Figure 4 shows the processing of a command, which outputs the literals `ab`, and repeats them twice. The *insert* part of the command is performed in step 1, and the rest shows the *copy* part step-by-step.

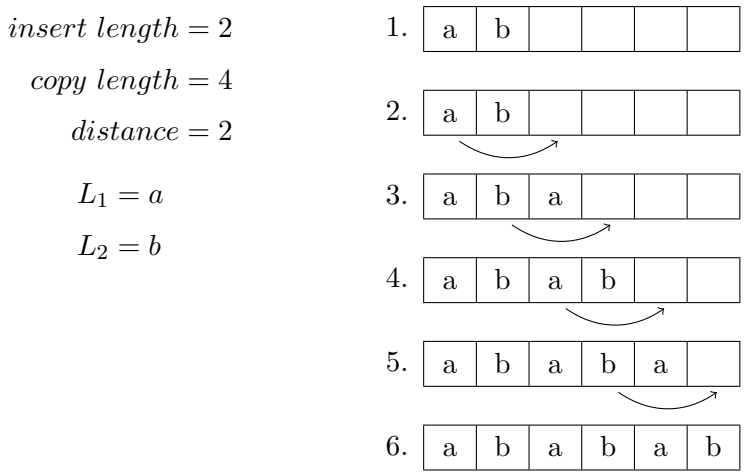


Figure 4: Processing of an insert&copy command that references output generated by itself.

3.6 Distance Codes and Parameters

The *distance* of an insert&copy command is written in the bit stream using a *distance code*. There are 3 types of distance codes:

- **Last code** refers to a ring buffer of the last 4 distances, optionally with an offset applied to the value. Brotli defines 16 such codes.
- **Complex code** refers to a range of distances. In the simplest case they work as intermediary codes, however they are more accurately described as finite arithmetic progressions whose common difference is set by the *postfix bits* parameter in the meta-block header.[5]
- **Direct code** refers to an exact distance between 1 and 121. The *direct code bits* parameter in the meta-block header determines whether direct codes are used, and sets the range of distances they cover. When in use, the ranges of all complex codes are offset so that any possible distance is covered by either a direct or complex code, but not both.

The per meta-block parameters allow optimizing for certain patterns of distances. A compression algorithm may prefer to search for sliding window references whose distances will follow these patterns, reducing the amount of complex codes in the meta-block.

Direct codes illustrate a trade-off — they can represent short distances without requiring any additional bits, but each unique distance must use a separate code, which will likely increase the size of the Huffman tree when many distinct distances are used.

[5] An example range $\{9, 10, 11, 12\}$ can exist when *postfix bits* is set to 0 (common difference of $2^0 = 1$). Increasing *postfix bits* to 1 makes the common difference $2^1 = 2$, and then one code refers to distances $\{9, 11, 13, 15\}$ and another code refers to distances $\{10, 12, 14, 16\}$.

3.7 Huffman Tree Selection

Insert&copy commands incorporate 3 categories of symbols represented using Huffman trees:

- `[L]` Literals
- `[I]` Insert&copy length codes
- `[D]` Distance codes

A simple meta-block may have one Huffman tree per category. In this section, we will look at the more interesting case — Brotli supports up to 256 distinct Huffman trees per category per meta-block. Each set of Huffman trees is defined in the header and understood as an array (fixed-size collection with 0-based indexing). When reading an element from any of the 3 categories, a decompressor must know which Huffman tree to choose from the array, and that is where two major features of Brotli — **block switching** and **context modeling** — come into play.

Keep in mind that the abbreviations `L,I,D` will often appear in the context of insert&copy commands, and the two mentioned features.

3.7.1 Block-Switch Commands

The 3 categories can be individually partitioned into blocks of varying lengths. Each block is assigned a non-unique *block type*. A meta-block may define up to 256 block types per category.

During decompression, the meta-block keeps track of each category's current block type and counter. Every time a Huffman tree for one of the categories is about to be used, its counter is checked. If the counter equals 0, a block-switch command is immediately read from the bit stream, which updates the current block type & counter. Finally, the counter is decremented. At the beginning of a meta-block, each category has its block type initialized to 0, and its initial counter value is part of the meta-block header.

In the bit stream, a block-switch command is made of a *block type code* followed by an intermediary *block length code*. The former will be explored in sections **??** and **??**.

In case of insert&copy command length codes, the block type is simply an index in the Huffman tree array. Figure 5 shows a meta-block with an array of 3 Huffman trees for length codes $HTREE_I$, thus also 3 block types BT_I, and a short sequence of insert&copy commands IC interleaved by block-switch commands. The counter BC_I was initialized to 4. Underneath, you can see which Huffman tree T is used for each insert&copy command.

$$HTREE_I \;=\; [\,T_0, T_1, T_2\,]$$

IC_1	IC_2	IC_3	IC_4	$BT_I = 2$ $BC_I = 1$	IC_5	$BT_I = 0$ $BC_I = 2$	IC_6	IC_7	$BT_I = 1$ $BC_I = \ldots$	IC_8	$\ldots$
T_0					T_2		T_0			T_1	

Figure 5: Sequence of insert&copy commands with interleaved block-switch commands.

Figure 6 shows contents of 2 insert&copy commands IC (literals L followed by distance D) and tracks the block type for literals BT_L. The counter BC_L was initialized to 3. The figure illustrates separation of categories, as BT_L retains its value when crossing command boundaries.

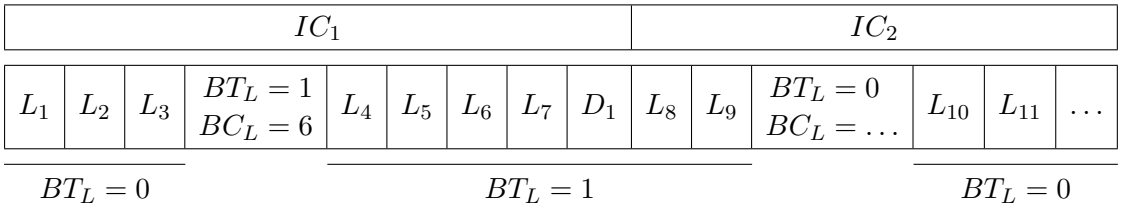


Figure 6: Tracking block type for literals across multiple insert&copy commands.

When it comes to literals and distance codes, the relationship between block types and Huffman tree indices is slightly more complicated as it uses *context modeling*.

3.7.2 Context Modeling

All block types for literals and distance codes are further subdivided into fixed-size groups. The groups are identified by zero-based *context IDs*. A *context map* is a surjective function mapping every possible $\langle$ *block type*, *context ID* $\rangle$ pair to an index of the appropriate Huffman tree:

- Literal context map has 64 context IDs per block type
- Distance context map has 4 context IDs per block type

The mapping can be implemented as an array with (*block types* $\times$ *context IDs per block type*) bytes. One byte is enough to store the index of any of the 256 possible Huffman trees.

Figure 7 depicts a possible distance context map in a meta-block with 7 Huffman trees and 3 block types for distance codes. The example visually separates the block types for clarity.

$$HTREE_D \;=\; [\,T_0,\, T_1,\, T_2,\, T_3,\, T_4,\, T_5,\, T_6\,]$$

$$CMAP_D \;=\; \underbrace{\boxed{0}\boxed{0}\boxed{0}\boxed{2}}_{BT=0} \cdot \overbrace{\underbrace{\boxed{1}\boxed{1}\boxed{2}\boxed{2}}_{BT=1}}^{\text{context ID} = 3} \cdot \underbrace{\boxed{3}\boxed{4}\boxed{5}\boxed{6}}_{BT=2}$$

Figure 7: Example of a distance context map.

The highlighted item (*block type* $=$ 1 $\wedge$ *context ID* $=$ 3) is the $(4 \times 1 + 3)$-th entry in the context map array. The value at that position is 2, corresponding to the Huffman tree T_2.

3.7.2.1 Literal Context ID

For literals, the context ID is calculated from the 2 most recently output bytes. Models that use 2 previous bytes for context are referred to as order-2 models[6].

Brotli processes the previous 2 bytes using a surjective function called *context mode*. The format defines 4 such functions, namely `LSB6`, `MSB6`, `UTF8`, and `Signed`.[6] A context mode maps all 2-byte combinations (2^{16} possibilities) onto the 64 context ID values.

Every block type within a meta-block can set its own context mode. In practice, the official compressor — in its current form — only uses 1 context mode for all block types in a meta-block. Furthermore, it only considers 2 out of the 4 defined context modes based on a heuristic.

3.7.2.2 Distance Context ID

For distance codes, the context ID depends on the value of *copy length*:

$$context\ ID = \begin{cases} 0, & \text{if } copy\ length = 2 \\ 1, & \text{if } copy\ length = 3 \\ 2, & \text{if } copy\ length = 4 \\ 3, & \text{if } copy\ length \geq 5 \end{cases}$$

Assigning context IDs in this way *might* be based on the assumption that short backward references are more likely to be found in short distances. It also isolates Huffman trees with potentially very long distances referring to dictionary words, as there are no dictionary words of length 2 or 3 (context IDs 0 and 1).

A full analysis is out of scope for this project, but we can at least look at the distribution of distance context IDs in the test data corpus. Do keep in mind that only quality levels 10 and 11 can take advantage of distance context IDs, but other quality levels are included out of interest.

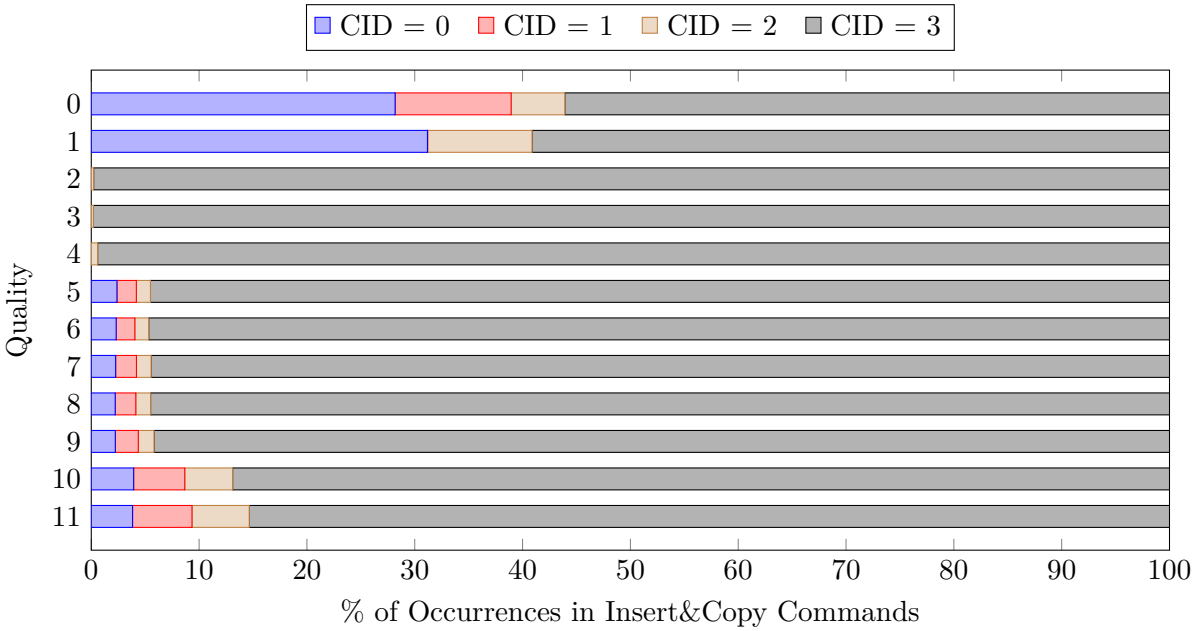


Figure 8: Distributions of distance context IDs in the test corpus.

[6] Technically, `LSB6` and `MSB6` are order-1 context models as they ignore the second-to-last byte.

4 Official Implementation

This section focuses on the official compressor implementation. We will begin by summarizing differences between the quality levels (0–11). Then, we will delve into individual features, exploring their implementation and their effect on the test corpus.

4.1 Official Quality Levels

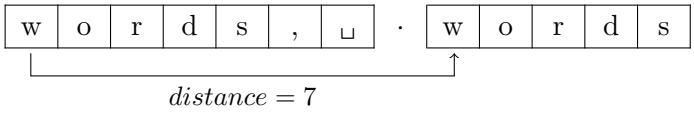
4.1.1 Quality Levels 0–1

The two lowest quality levels are reimplementations of Google's high speed Snappy[7] compressor adapted to the Brotli format. Both qualities generate insert&copy commands in an interesting way that reduces the amount of length codes defined in the header — instead of generating one command with both the *insert* and *copy* part, it splits them into two commands.

The first command is for the *insert* part, and because every command must have a minimum copy length of 2, it also includes 2 bytes of the copy. The second command outputs the rest of the copy, using distance code zero as the distance is the same as that of the previous command.

We can visualize the commands with an example. Figure 9 shows how one command generating the phrase "words, words" would be split into two.

1. *insert length* $= 7$, *copy length* $= 5$, *distance* $= 7$



1. *insert length* $= 7$, *copy length* $= 2$, *distance* $= 7$
2. *insert length* $= 0$, *copy length* $= 3$, *distance* $= 7$ (always the same distance)

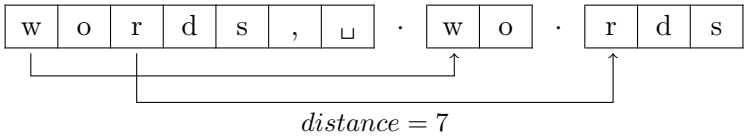


Figure 9: Insert&copy command generation pattern in official compressor's lowest quality levels.

Instead of 704 length codes covering all possible combinations of *insert* and *copy* codes and `IDCZ` marks, we merely have to account for the following cases covered by only 59 length codes:

- (*insert length* > 0 $\wedge$ *copy length* $= 2$)
- (*insert length* $= 0$ $\wedge$ *copy length* ≥ 2)
- (*insert length* $= 0$ $\wedge$ *copy length* ≥ 2 $\wedge$ `IDCZ`)

[7] `https://github.com/google/snappy`

This simplifies meta-block generation in both quality levels. Quality level 0 starts with predefined length & distance code Huffman trees, which means those trees must include all possible codes, and it immediately begins outputting commands. If a second (third, fourth, etc.) meta-block is generated, it will use knowledge about the previous meta-block codes to adjust the length & distance code trees. Quality level 1 and higher generate commands for a meta-block first, and construct trees based on the actual encoded data.

4.1.2 Quality Levels 2–9

All middle quality levels are fundamentally similar. They use a variety of hash structures and strategies to find backward references, all tuned for different quality levels, window sizes, and input sizes. With increasing quality level, Brotli starts enabling additional features:

- **Quality levels 2 and 4–9** use a simplified, limited version of the static dictionary and word transform system.[8]
- **Quality level 4** and higher perform the previously mentioned optimization that modifies some Huffman trees to make their path lengths form longer runs.
- **Quality level 4** is also the first to perform block splitting. All context maps follow the same pattern, in which every block type has its own distinct Huffman tree.
- **Quality level 5** enables basic context modeling for literals using the `UTF8` literal context mode, heuristically choosing from 3 predefined context map patterns.

The middle quality levels begin using most of the advanced features of Brotli, however most of them use a completely different — simpler but faster — approach than the highest quality levels.

It is worth paying attention to middle quality levels, because they are used as default settings in HTTP server software (level 5 in Apache[9] and level 6 in NGINX[10]), as they can compress dynamic web content reasonably fast while the highest quality levels may only be suitable for static content.

4.1.3 Quality Levels 10–11

The two highest quality levels are based on techniques used in Google's older Zopfli[11] compressor, combined with much more sophisticated use of the Brotli format features compared to previous quality levels.

[8] In quality level 4, the static dictionary is only enabled for files smaller than `1 MiB`.
[9] `https://httpd.apache.org/docs/2.4/mod/mod_brotli.html`
[10] `https://github.com/google/ngx_brotli`
[11] `https://github.com/google/zopfli`

The differences between the two quality levels are in tuning of the hash structure and Zopfli parameters, and spending more time refining block splits. Compared to previous quality levels though, the differences in advanced features are much more significant:

- Block splitting uses a completely different algorithm
- Literal & distance context maps are generated based on the insert&copy commands
- Distance parameters are selected by the compressor based on the insert&copy commands
- More thorough use of the static dictionary and its transform system

4.2 Feature Evaluation

We will now look at the official implementation(s) of important features across different quality levels, starting with the simplest features and working our way up. We will try turning each individual feature off to see what effect that has on the test corpus. Footnotes will point to important source code files and relevant functions.

4.2.1 Evaluating Huffman Tree Run Optimization

This optimization, which is enabled for quality levels 4 and higher, modifies Huffman trees for literals, length codes, and distance codes to make their path lengths form longer runs. It reduces how many bits the trees use in the header, but increases how many bits the *symbols* use in the data section. In most cases, the savings are more significant than the losses — in the test corpus, the optimization on average saved 564 bytes and lost 376 bytes per file.

Turning the feature off increased compressed sizes by an average of ≈ 0.3% per file. On the other hand, it also ended up reducing sizes of 63 files (out of 1352 files compressed using quality levels 4–11) by ≈ 0.03% per file. Figure 10 shows the difference in total sizes.

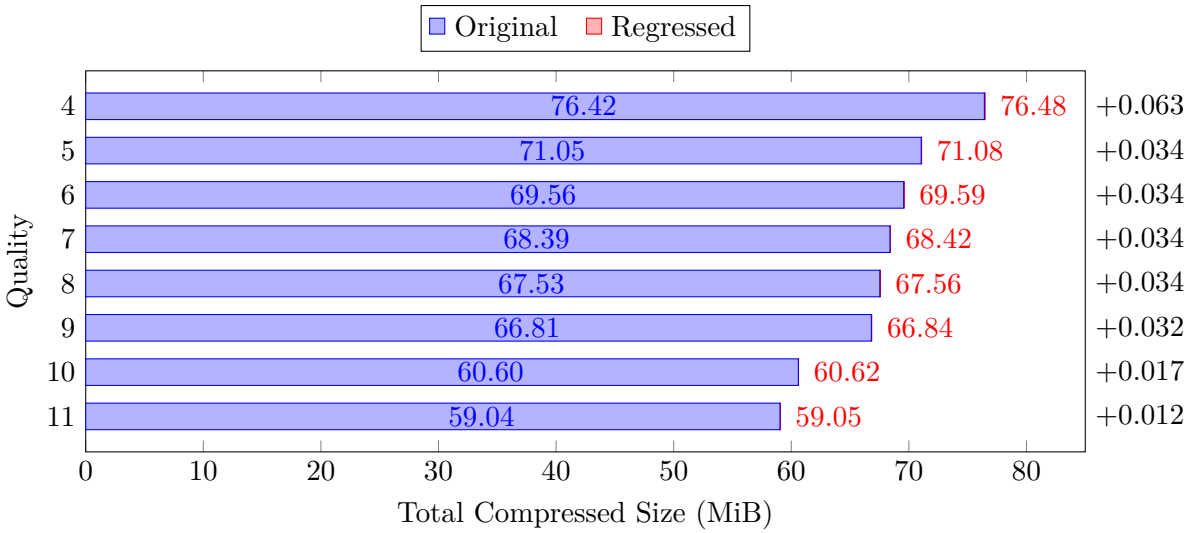


Figure 10: Test corpus size after disabling Huffman tree run optimization.

4.2.2 Evaluating Distance Parameters

The *postfix bits* parameter has a value of 0–3 and *direct code bits* has a value of 0–15,[12] totaling $4 \times 16 = 64$ possible configurations. The default value of both is zero.

When compressing with quality levels 10 and 11, the compressor tests a heuristically chosen subset of configurations, and chooses the one for which it estimates the smallest footprint.[13] We can make several observations about distance parameters across the test corpus:

- Out of 169 files, the amount which had one or more meta-blocks with at least one non-zero distance parameter was 72 for quality 10, and 52 for quality 11.
- Out of the total 360 meta-blocks in both qualities, 145 had at least one non-zero parameter.
- The feature seems more likely to be used for large files. The average uncompressed file size is ≈ `3.5 MiB` overall, but ≈ `7.2 MiB` when only counting files which use distance parameters. In the Silesia corpus assembled from files ranging from `5 MiB` up to `50 MiB`, only 1 file compressed with quality 11 did not use the feature.

Turning the feature off increased compressed sizes by an average of ≈ 0.08% and ≈ 0.10% per file for quality 10 and 11 respectively. On the other hand, it also:

- Reduced sizes of 19 files using quality 10, by ≈ 0.032% per file
- Reduced sizes of 11 files using quality 11, by ≈ 0.038% per file

Figure 11 shows the total compressed sizes, and the change after the feature was turned off.

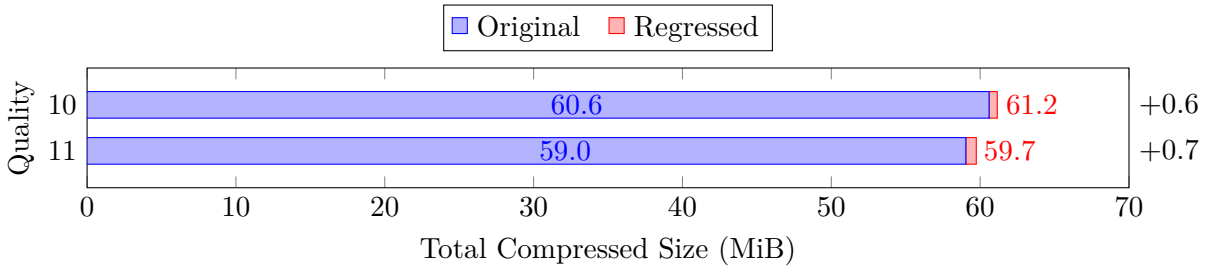


Figure 11: Test corpus size after disabling distance parameters.

4.2.3 Evaluating Static Dictionary

- **Medium quality levels (2–9)**[14]
  - Lookups are performed only if a backward reference search fails. If too many lookups fail, the compressor will stop checking the dictionary for the rest of the file.

[12] The amount of **direct** distance codes is calculated as $\left(\textit{direct code bits} \times 2^{(\textit{postfix bits})}\right)$, allowing for up to 120 codes that directly represent distances 1–121.

[13] Source file `metablock.c` (`BrotliBuildMetaBlock`).

[14] Source files `dictionary_hash.c`, `hash.h` (`SearchInStaticDictionary`).

  - Words are looked up using a simplified hash table, in which every bucket can hold at most 2 words.
  - Additionally, the words are organized so that the first word has a length of 8–24, and the second word has a length of 4–7. Quality levels 2–4 check only the first word.
  - Due to hash collisions and the bucket limit, the table contains only 6 031 words out of the 13 504 defined in the dictionary.
  - The transform system is limited to 10 transforms — those which have no prefixes and suffixes, and use the **Identity** or **Omit Last 1–9** functions.

- **High quality levels (10–11)**[15]
  - Lookups are performed alongside all backward reference searches.
  - Words are looked up using a hash table containing all 13 504 words.
  - The transform system is used to nearly full extent, however 8 of the 121 possible transforms — those based on the **Omit First 1–9** functions — are unused.[16]

We can confirm these findings by counting how many times the transform functions were used in each quality level. The numbers are presented in table 2, with sums on the bottom showing us how many dictionary references each quality level has produced in total.

Table 2: Dictionary transform function usage across the test corpus.

Function \ Quality	2	4	5	6	7	8	9	10	11
Identity	770	5 860	28 954	27 614	26 899	26 703	26 233	90 239	98 464
Ferment First	0	0	0	0	0	0	0	14 426	19 532
Ferment All	0	0	0	0	0	0	0	7 228	9 116
Omit Last 1	239	2 106	5 362	5 153	5 047	4 999	4 931	3 012	4 355
Omit Last 2	377	2 001	2 263	2 129	2 067	2 053	1 992	921	1 544
Omit Last 3	399	2 991	1 450	1 325	1 286	1 265	1 217	1 057	2 055
Omit Last 4	255	2 269	1 287	1 184	1 141	1 123	1 103	528	1 053
Omit Last 5	119	1 861	724	680	664	664	645	204	368
Omit Last 6	48	1 347	548	523	504	503	496	196	393
Omit Last 7	50	865	444	435	426	422	412	244	397
Omit Last 8	107	872	346	331	327	324	315	225	377
Omit Last 9	41	410	277	256	251	246	242	239	423
Omit First 1–9	0	0	0	0	0	0	0	0	0
	2 405	20 582	41 655	39 630	38 612	38 302	37 586	118 519	138 077

[15] Source files `dictionary.c`, `static_dict_lut.h`, `static_dict.c`, `hash_to_binary_tree_inc.h`.

[16] Curiously, **Omit First 8** is not used in any of the 121 transforms defined in the format, which explains why there are only 8 transforms based on the 9 functions.

Due to how the default dictionary was constructed, we might expect it to have a disproportionate effect on texts in certain spoken and computer languages. Unfortunately, even amidst files with similar kind of content, there is too much variance to draw a firm conclusion — to demonstrate this, table 3 shows the % of bytes generated by dictionary references in English plain text files.

Table 3: Demonstration of varying dictionary usage in English plain text files compressed with the highest quality level (11).

File	Uncompressed Size	Dictionary-Generated
`Canterbury/lcet10.txt`	`416 KiB`	14.17%
`Canterbury/plrabn12.txt`	`471 KiB`	7.25%
`Silesia/dickens.txt`	`9.72 MiB`	1.28%
`Bible/English.txt`	`4.05 MiB`	0.85%

To get a better idea of how efficient the dictionary is, we can try turning it off. We should keep mind that the effect will be suppressed, because the compressor may find backward references near words that would have otherwise become dictionary references, and avoiding large distance codes helps also.

Figure 12 shows the total sizes. Despite the already sparse use of the dictionary in quality level 2, turning it off reduces the total size. The reduction can be attributed to large files in particular, as the results suggest that the dictionary in quality level 2 tends to benefit small files and negatively affect large files.

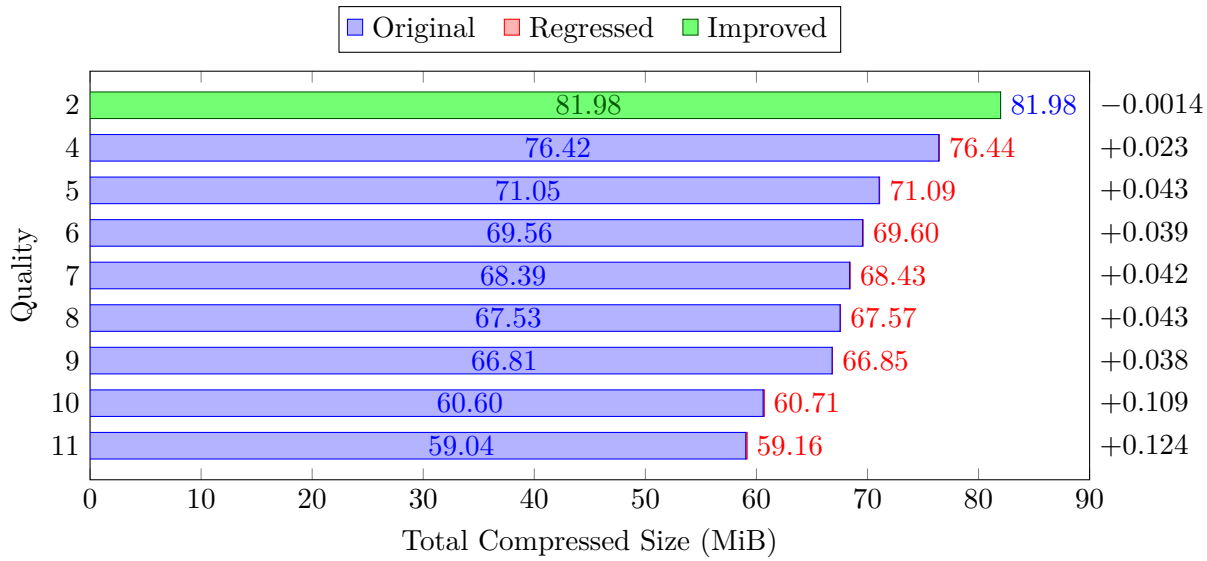


Figure 12: Test corpus size after disabling the static dictionary.

We can notice that the deltas across quality levels 5–9 and 10-11 are similar, so the next part will only consider levels 5 and 11.

Turning the dictionary off for quality levels 5 and 11 resulted in a total size increase of $\approx 0.06\%$ and $\approx 0.21\%$ respectively. Looking only at the sub-corpus of downloaded website resources, we find more dramatic increases of $\approx 0.72\%$ and $\approx 1.81\%$. Regardless of the exact reason, we can conclude that the dictionary does have a significant effect, especially on files falling under the intended use case.

4.2.4 Evaluating Block Splitting & Context Modeling

Starting with quality level 4, block splitting is done for all 3 categories of symbols. The official compressor uses two different approaches for both block splitting and context modeling.

4.2.4.1 Medium Quality Levels (4–9)

Block splitting uses a *greedy* algorithm. It performs a single pass over symbols in each category, periodically deciding whether to add a block-switch command that refers to either a completely new block type, or the second-to-last block type.[17]

Context modeling is used for literals starting with quality level 5, but never used for distance codes. Only the `UTF8` literal context mode is used.[18]

If the compressor decides to use context modeling, it heuristically picks one of 3 preset context map patterns.[19] Table 4 shows how many times each pattern emerged in meta-blocks in the test corpus.

Table 4: Preset context map pattern usage across the test corpus.

Quality	No Context Model	2-Tree Pattern	3-Tree Pattern	13-Tree Pattern
5	161	0	0	21
6	161	0	0	21
7	159	0	2	21
8	159	0	2	21
9	156	0	2	21

The context map pattern (or all zeros where context modeling is unused) for the first block type is repeated over all other block types, with each block type getting its own set of Huffman trees.[20] To ensure the amount of Huffman trees does not exceed 256, the block splitter is limited to $(256 \div \textit{trees per block type})$ block types.

[17] Source file `metablock.c` (`BrotliBuildMetaBlockGreedyInternal`).
[18] Source file `encode.c` (`ChooseContextMode`).
[19] Source file `encode.c` (`DecideOverLiteralContextModeling`).
[20] Same mechanism as `RepeatFirstBlockType(true)` in the context map builder API.

4.2.4.2 High Quality Levels (10–11)

The input file is analyzed early to determine which literal context mode to use — `UTF8` is used if at least ¾ of the input is deemed to be UTF-8 encoded, `Signed` mode is used otherwise.[21] Although the format supports 2 other literal context modes (`LSB6`, `MSB6`), and it can also set separate modes per block type, the official compressor uses neither feature at the time of writing.

After the insert&copy commands for a meta-block are generated, the 3 categories of symbols are split into separate sequences, and the block splitting algorithm is applied to each.[22]

Block splitting begins by pseudorandomly sampling the sequence into multiple histograms, which represent preliminary block types. Their amount depends on the length of the input sequence. Afterwards, the block splitter uses an iterative algorithm that works as follows:[23]

1. Assign each symbol to the histogram (block type) which encodes it most efficiently.
2. This creates an erratic pattern, adding a block-switch command at every change would be expensive. Instead, place marks at positions where a block switch would be desirable.
3. Starting from the end of the sequence, extend all blocks towards the beginning, so that block types only change at the marked positions.
4. Generate new histograms from the updated block types, and use them in the next iteration.

Quality level 10 performs 3 iterations, quality level 11 performs 10 iterations. Figure 13 shows an example of one iteration. Hollow circle marks the end of the sequence where step 3 begins, filled circles are at the marked switch positions decided by step 2.

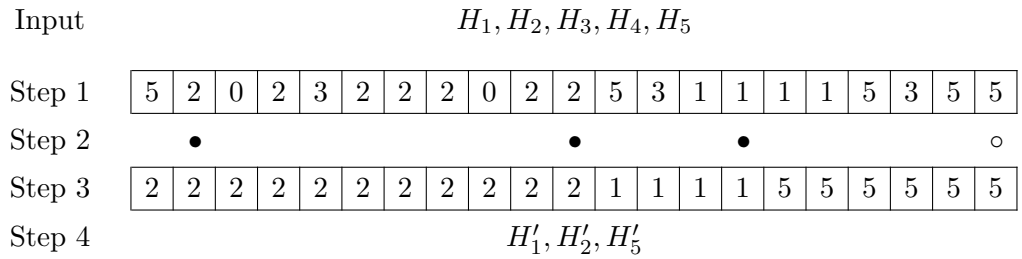


Figure 13: Example of one iteration of official compressor's advanced block splitting algorithm.

The resulting histograms go through a merging process, which reduces their amount by repeatedly finding 2 most similar histograms, and merging them into one.[24] Once their amount drops down to 256, merging may continue until there are no more pairs of "similar enough" histograms.

As the old block types may not perfectly match the new histograms, the final step goes over each block, creates a histogram of its symbol sub-sequence, compares that histogram to all of the new histograms, and the most similar one's block type gets assigned to that block. If adjacent blocks end up with the same block type, they will be combined.

[21] Source file `encode.c` (`ChooseContextMode`).
[22] Source file `block_splitter.c`.
[23] Source file `block_splitter_inc.h` (`FindBlocks`).
[24] Source file `block_splitter_inc.h` (`ClusterBlocks`).

Figure 14 shows an example of the merging process with 4 blocks B, and 3 block types BT correlated with 3 histograms H. One merge is performed ($H_1 \cup H_2 = H_{1,2}$) resulting in a new arrangement of 2 block types. Next, block types of the 4 blocks are reassigned to match the new arrangement, pointing out a possibility that blocks with the same initial block type (B_1, B_3) could end up matching different histograms. The adjacent blocks B_1 and B_2 are combined, as they end up with the same block type.

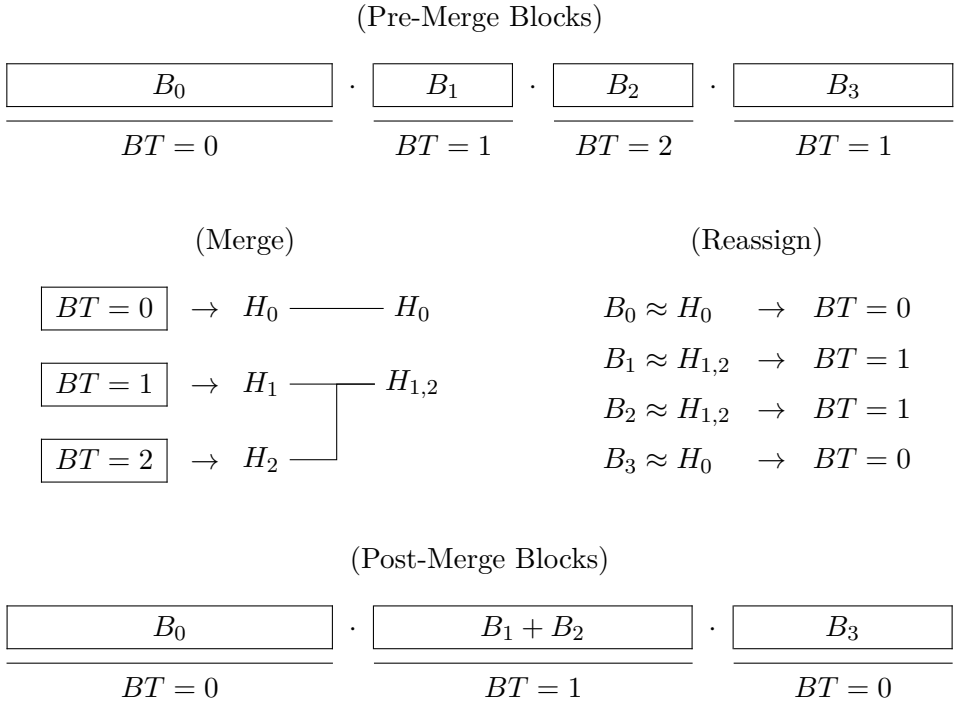


Figure 14: Example of how the block splitter could merge and reassign blocks generated by the previous step.

Block splitting is followed by context modeling. The algorithm first collects histograms of literals and distance codes for all possible $\langle$ *block type*, *context ID* $\rangle$ pairs — we can think of it as a context map where each pair maps to a unique Huffman tree. Of course, the format does not support unlimited trees, and every tree has a certain bit footprint, so the amount of trees is reduced with the same merging process used by the block splitter. Finally, the algorithm reassigns all context map indices — for each $\langle$ *block type*, *context ID* $\rangle$ pair, it compares the original histogram to all of the new histograms, and picks the most similar one.[25]

Figure 15 shows an example of how a distance context map with 1 block type could be created. It begins with 4 histograms H, which turn into 2 after the merging stage performs two merges ($H_1 \cup H_3 = H_{1,3}$ and $H_{1,3} \cup H_4 = H_{1,3,4}$). The right half shows results of the comparison between old and new histograms. The rightmost 4 squares are the final 4 context map values.

[25] Source file `cluster_inc.h`.

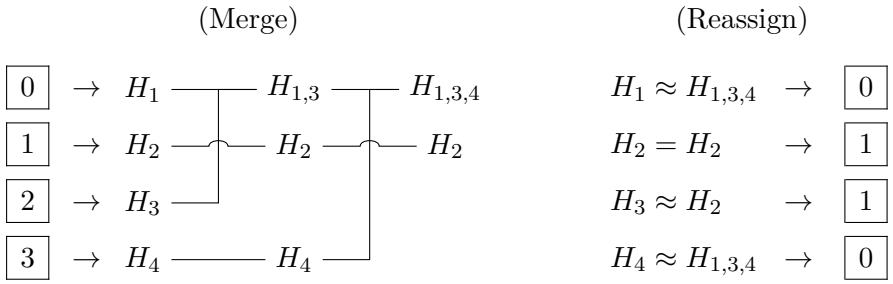


Figure 15: Example of how a distance context map could be created by the official compressor.

4.2.4.3 Evaluation

Figure 16 shows the total size difference after disabling context modeling, and instead assigning each block type a unique Huffman tree. Figure 17 turns off both features at once.

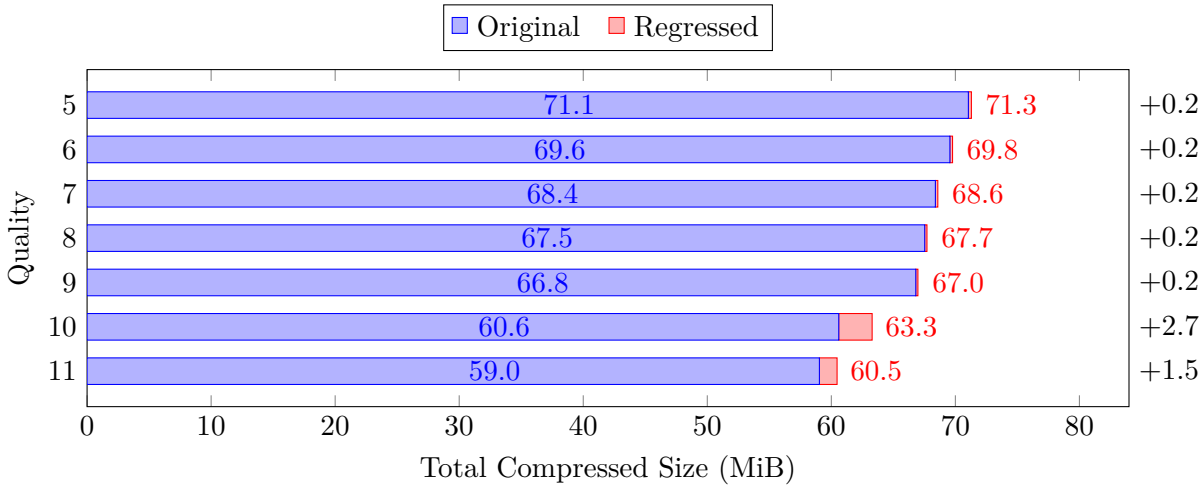


Figure 16: Test corpus size after disabling context modeling for both literals & distance codes.

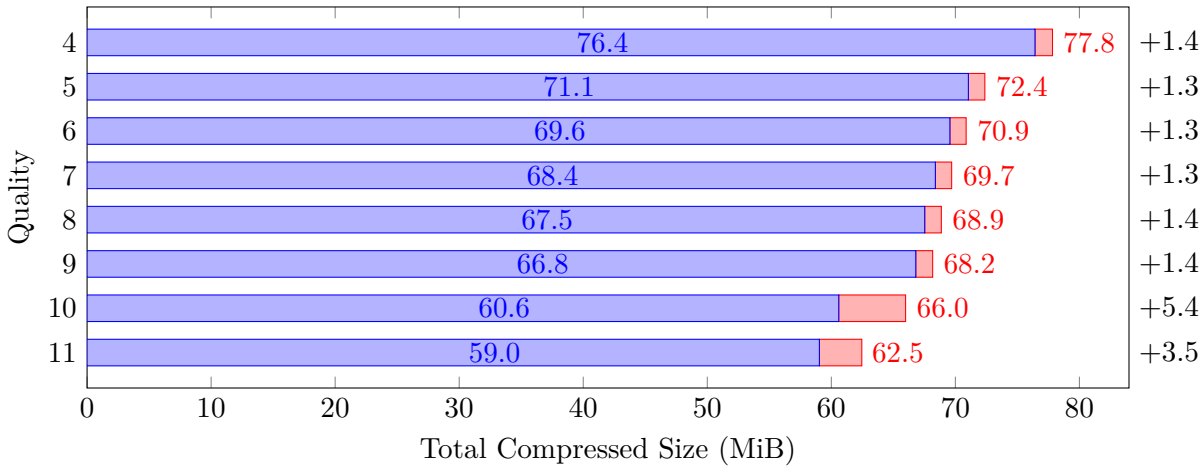


Figure 17: Test corpus size after disabling both block splitting and context modeling.

5 Conclusion

Brotli is a promising HTTP compression standard that delivers better overall results than other compression standards commonly used on the World Wide Web. Although the Brotli format specification[7] covers all information needed to implement a decompression algorithm, one of the goals of this thesis was to provide a more structured explanation with visual aids and examples, which should be compelling even to people with no previous knowledge of compression techniques.

The full thesis additionally covers a custom implementation of Brotli, and tests several experimental modifications to the official implementation. You can find the source code and full thesis at `https://github.com/chylex/Brotli-Builder`.

References

[1] P. Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*. RFC 1951. RFC Editor, 1996-05, pp. 1–17. URL: https://tools.ietf.org/html/rfc1951.

[2] David A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes". In: *Proceedings of the IRE* 40 (9 1952-09), pp. 1098–1101. ISSN: 2162-6634. DOI: 10.1109/JRPROC.1952.273898.

[3] Jacob Ziv and Abraham Lempel. "A universal algorithm for sequential data compression". In: *IEEE TRANSACTIONS ON INFORMATION THEORY* 23.3 (1977), pp. 337–343.

[4] David Salomon. *Data Compression: The Complete Reference*. With contributions by Giovanni Motta and David Bryant. Berlin, Germany / Heidelberg, Germany / London, UK / etc.: Springer-Verlag, 2007. xxvii+1092. ISBN: 1-84628-602-6.

[5] Jyrki Alakuijala et al. *Comparison of Brotli, Deflate, Zopfli, LZMA, LZHAM and Bzip2 Compression Algorithms*. 2015-09-22. URL: https://github.com/google/brotli/blob/master/docs/brotli-comparison-study-2015-09-22.pdf.

[6] Dario Phong. *Finite Context Modelling*. 2000-01-02. URL: http://www.hugi.scene.org/online/coding/hugi%2019%20-%20cofinite.htm.

[7] J. Alakuijala and Z. Szabadka. *Brotli Compressed Data Format*. RFC 7932. RFC Editor, 2016-07, pp. 1–128. URL: https://tools.ietf.org/html/rfc7932.