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

Brotli Compression Algorithm

2020 Daniel Chýlek

Foreword

You are reading an online version of my masters thesis. The source codes and issue tracker are available at `https://github.com/chylex/Brotli-Builder`.

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

Abstract

This thesis is a comprehensive exploration of the Brotli compression algorithm and data format. After explaining key principles Brotli is built upon, the paper introduces a custom implementation that provides tools to study the format and develop new format-compatible compression techniques. The custom implementation is followed by an in-depth look at the official compressor implementation, and how different quality levels utilize features of the format. The paper concludes by using the gained insight to experiment with the format and compression techniques.

Keywords: lossless compression; Brotli; Huffman coding; context modeling

Contents

List of Symbols and Abbreviations

API	–	Application Programming Interface
CLI	–	Command Line Interface
CRLF	–	Carriage Return + Line Feed
CSS	–	Cascading Style Sheets
GUI	–	Graphical User Interface
HTML	–	Hypertext Markup Language
HTTP	–	Hypertext Transfer Protocol
JS	–	JavaScript
LZ77	–	Lempel-Ziv 77
MSVC	–	Microsoft Visual C++
RFC	–	Request for Comments
SDK	–	Software Development Kit
UTF	–	Unicode Transformation Format
␣	–	Indicates a single space character
`B`	–	Byte (8 bits)
`KiB`	–	Kibibyte, `1 KiB` = `1024 B`
`MiB`	–	Mebibyte, `1 MiB` = `1024 KiB`
`GiB`	–	Gibibyte, `1 GiB` = `1024 MiB`

List of Figures

List of Tables

Listings

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.

This thesis introduces a program library, which implements Brotli compression and decompression based on the RFC7932[2] specification, as well as several utility applications intended to aid understanding and analysis of Brotli compressed files. This is followed by an in-depth exploration of the official implementation, and the differences between it and the custom implementation. The insight gained by studying the format is used to propose and test several experimental modifications to the official implementation, which intend to improve compression while maintaining format compatibility.

Section 2 describes the organization of the programming projects, information about the software setup, and the compression corpus used for testing and validation. The section ends with 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 talks about the technical background of the main program library, decisions that went into the custom implementation, and examples of how the library API can be used.

Section 5 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. The third part concludes with several experiments that modify the official source code in an attempt to find improvements.

2 Setup & Organization

2.1 Project Organization

The custom implementation and utilities were written in `C# 8`, and organized into several projects in a Visual Studio 2019 solution:

Name	Type	Framework	Description
`Brotli Lib`	Library	`.NET Standard 2.1`	Main library — implementation of Brotli
`Brotli Impl`	Library	`.NET Standard 2.1`	Example uses of the main library API
`Brotli Builder`	GUI[1]	`.NET Core 3.0`	Utility for file analysis & comparison
`Brotli Calc`	CLI	`.NET Core 3.0`	Utility for batch file processing

All projects are available at `https://github.com/chylex/Brotli-Builder`.

2.2 Brotli Compilation

The Brotli executable and all its modified versions were built using the official repository at `https://github.com/google/brotli`, and based on the `c435f06` commit from October 1, 2019. This version included a few improvements committed after the official v1.0.7 release. The following software configuration was used for all builds:

- `Visual Studio 2019 (16.3.9)`
- `LLVM 8.0.1`
- `MSVC 14.23.28105`
- `Windows SDK 10.0.18362.0`
- `clang_cl_x64` toolset
- `Release` configuration

2.3 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.

[1] The GUI is based on Windows Forms, which requires `Desktop Runtime` to be installed alongside `.NET Core`.

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] (2.7 MiB)
- The Silesia Corpus[3] (202 MiB)
- A selection of files from Snappy Test Data[4] (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[5], Chinese (Simplified)[6], Chinese (Traditional)[7], English[8], Hindi[9], Russian[10], Spanish[11]
  - Each language was downloaded in the *Plain text canon only chapter files* format. Text files from each archive were combined with a new line (CRLF) following each file.
- HTML, CSS, and JS files from several of the most popular websites (20.4 MiB)
  - baidu.com, facebook.com, google.com, instagram.com, vk.com, wikipedia.org, yandex.ru, youtube.com
  - Each website was downloaded using the *Save page as...*, *Web Page, complete* feature in an anonymous window in Firefox.

2.3.1 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.

[2] http://corpus.canterbury.ac.nz/descriptions/#cantrbry
[3] http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia
[4] https://github.com/google/snappy/tree/e9e11b84e629c3e06fbaa4f0a86de02ceb9d6992/testdata
[5] https://ebible.org/find/details.php?id=arbnav
[6] https://ebible.org/find/details.php?id=cmn-cu89s
[7] https://ebible.org/find/details.php?id=cmn-cu89t
[8] https://ebible.org/find/details.php?id=eng-asv
[9] https://ebible.org/find/details.php?id=hin2017
[10] https://ebible.org/find/details.php?id=russyn
[11] https://ebible.org/find/details.php?id=spaRV1909

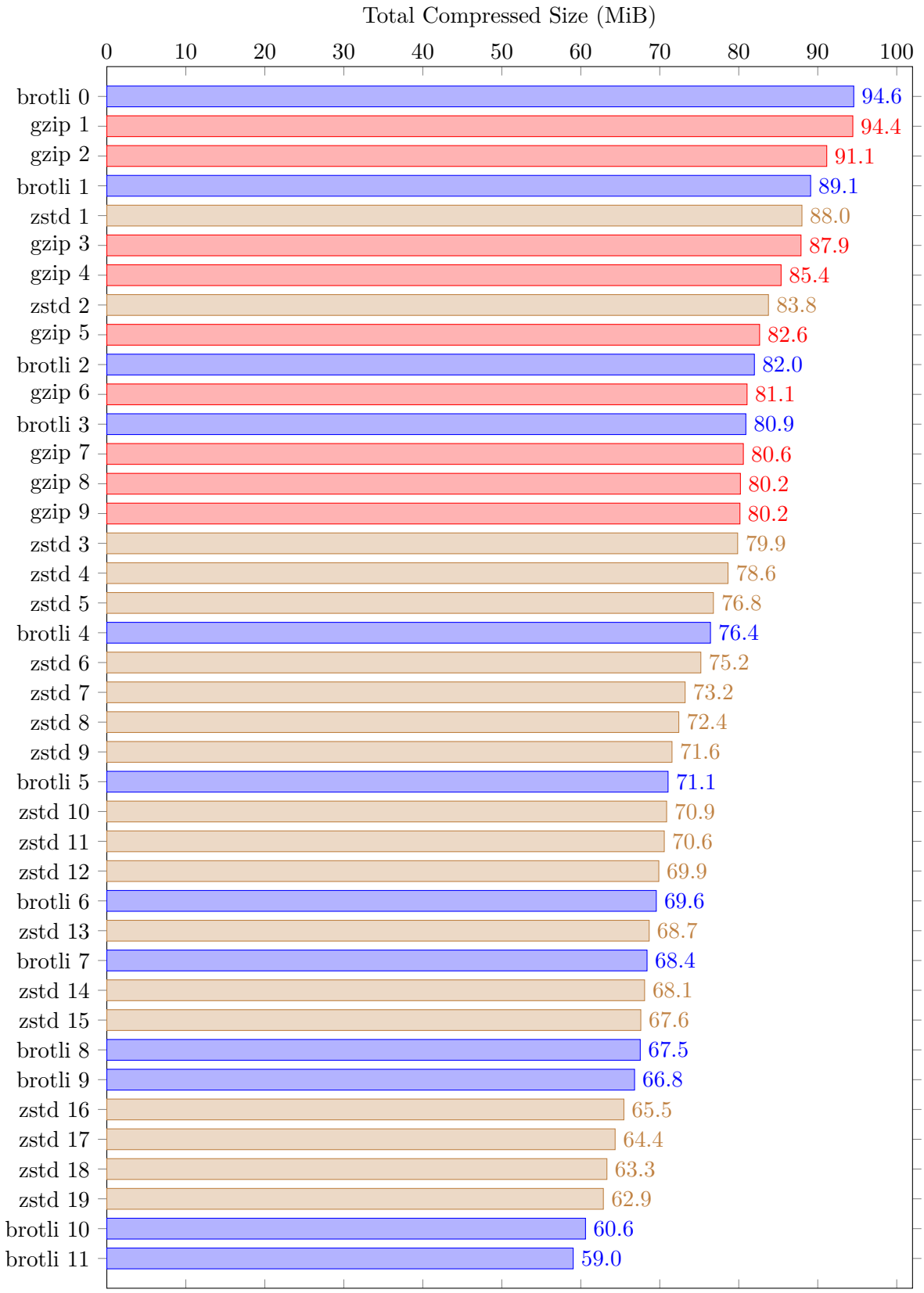


Figure 1: Test corpus size comparison between Brotli, gzip, and zstd.

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.

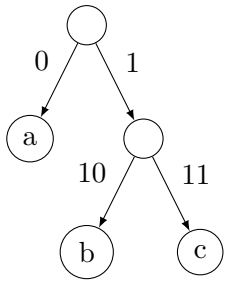
3.1 Huffman Coding

Binary Huffman coding encodes *symbols* of an *alphabet* using variable-length bit sequences[3]. 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[4], 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.[12]

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.[5]

[12] `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.[13] 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.[14]

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[15] 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.

[13] 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[6].

[14] 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.

[15] 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 4.4.1 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.

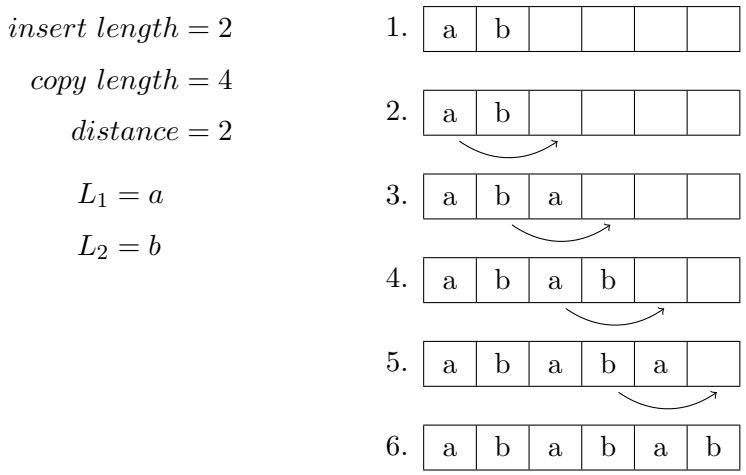


Figure 4: Processing of an insert&copy command that references output generated by itself.

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.[16]
- **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.

[16] 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 4.4.2 and 4.5.3.

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.

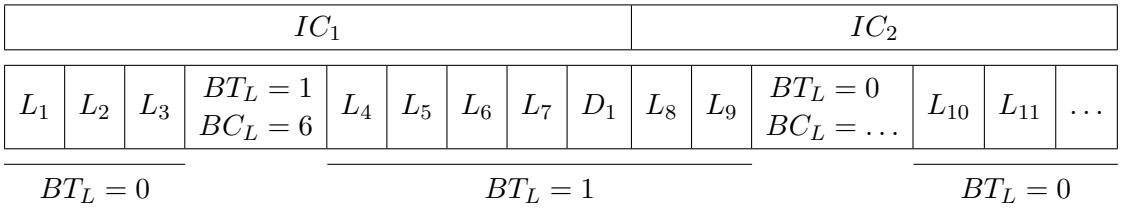


Figure 6: Tracking block type for literals across multiple insert&copy commands.

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[7].

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`.[17] 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.

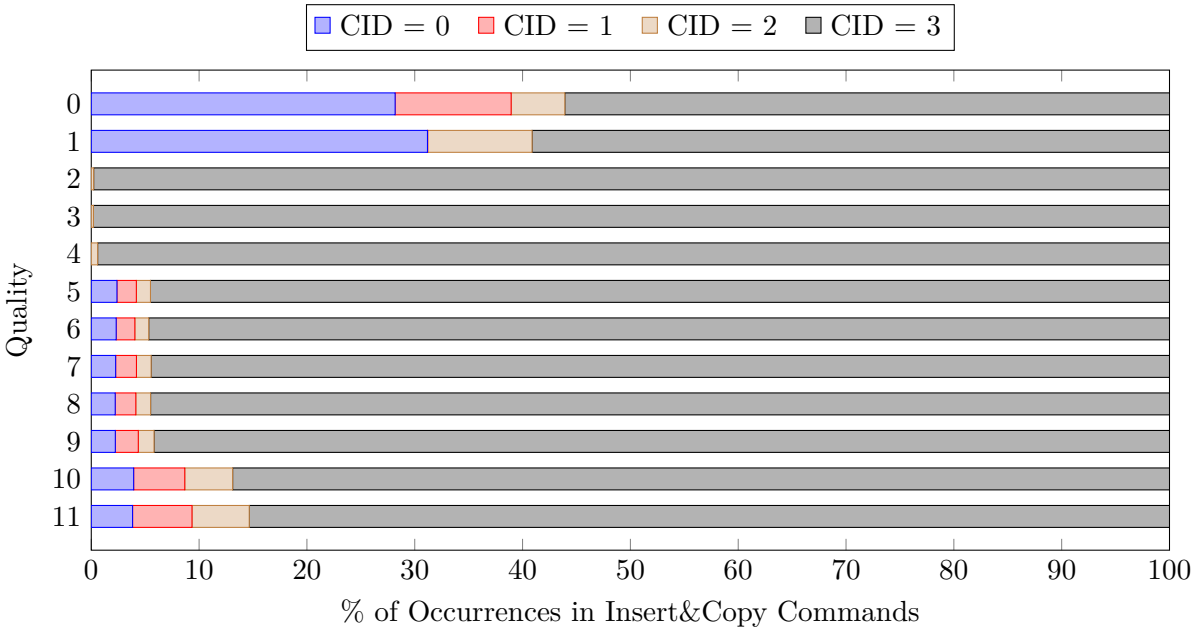


Figure 8: Distributions of distance context IDs in the test corpus.

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

4 Implementing Brotli

The custom implementation resides in the `BrotliLib` program library. It represents the format as a composition of classes we will collectively call *component classes*. A Brotli compressed file is represented by a `BrotliFileStructure` object. The core of the library API revolves around operations with this object, with streaming APIs available in cases when we do not need the whole structure to be loaded into memory.

Working with a Brotli file requires tracking state that transcends meta-block boundaries, such as the sliding window buffer, static dictionary, or recently used literals and distances. This state is stored in a `BrotliGlobalState` object, which also includes logic for resolving backward references and dictionary references, and can be used to obtain whole decompressed output.

In the context of `BrotliFileStructure` operations, `byte[]` denotes an array of uncompressed byte data, and `BitStream` is a container providing access to individual bits, which is necessary to work with compressed files. First, let us define 3 basic operations:

Deserialization `BitStream` → `BrotliFileStructure`

Converts a Brotli compressed file into its object representation.

Decompression `BrotliFileStructure` → `byte[]`

Extracts uncompressed data out of an object representation.

Serialization `BrotliFileStructure` → `BitStream`

Converts an object representation into a Brotli compressed file.

These operations are enough to read/write `BrotliFileStructure` objects from/to files. The problem is that although all component classes are exposed to users of the library, constructing an entire structure from scratch is unwieldy due to (1) requiring deep knowledge of both the Brotli specification and component classes to construct it in a valid way, and (2) having to write boilerplate code, especially for meta-block headers.

To address both issues, the library includes builder-style APIs that guide users through the process, validate the data, and provide means to implement the following operations:

Encoding `byte[]` → `MetaBlock`

Generates one meta-block from uncompressed bytes. Called repeatedly until all bytes are consumed, generating as many meta-blocks as necessary.

Transforming `MetaBlock` → `MetaBlock[]`

Transforms one meta-block into any amount of other meta-blocks. The definition is flexible, in that it allows converting one meta-block into another, splitting a meta-block into multiple, or removing a meta-block altogether — as long as the resulting structure remains valid.

The generator-style operations let us compose them in a pipeline made of 1 encoder and M transformers. The diagram in figure 9 shows an encoder generating meta-blocks MB_O, a transformer chain $T_1 \dots T_m$ producing transformed meta-blocks MB_T, and the output structure containing final transformed meta-blocks.

We must be careful about handling `BrotliGlobalState` denoted S. At every step — row in the diagram — the encoder and all transformers start with a copy of the previous final meta-block's output state. For example, MB_{O_2} and all $MB_{T_{2,*}}$ are given the state at the end of $MB_{T_{1,m}}$, and so the dashed line shows state S_{T_1} from $MB_{T_{1,m}}$ being fed back into the encoder.

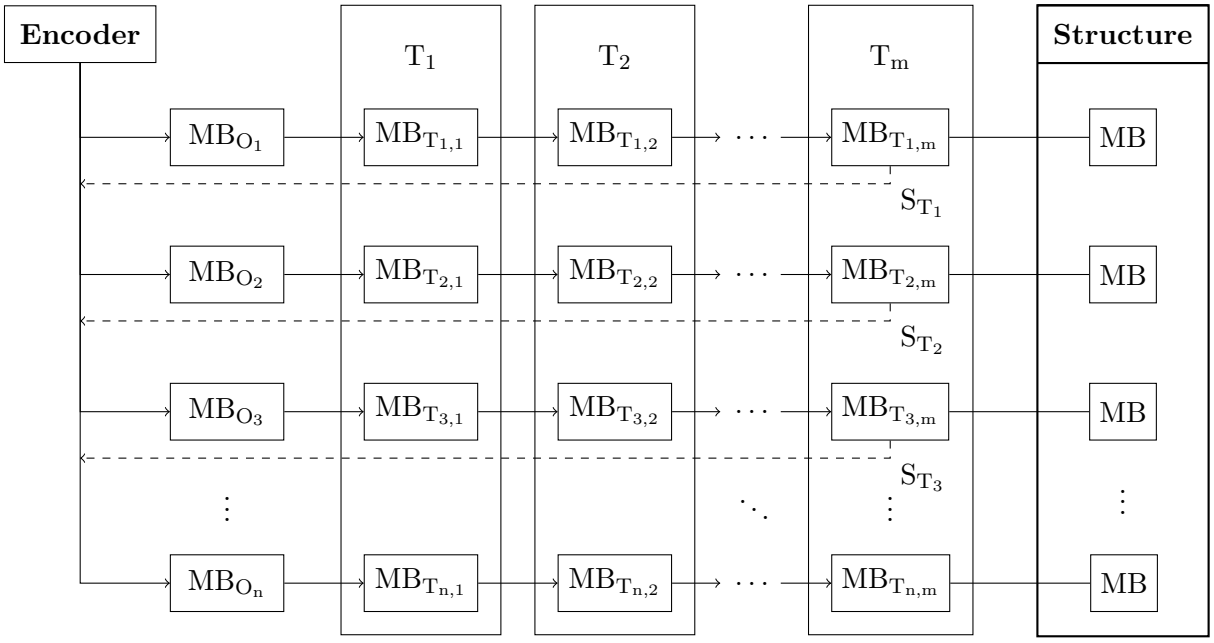


Figure 9: Visualization of the encode-transform pipeline.

This pipeline is implemented under the name `BrotliEncodePipeline`. Transformers can also be applied to a whole `BrotliFileStructure` or individual meta-blocks in the streaming API, if we wish to edit a file we did not create ourselves.

Before we proceed further, let us contemplate the library design. Firstly, the object representation trades off memory efficiency and construction overhead for something easy to understand and manipulate. Secondly, the fine-grained separation of operations and pipeline design allow for transformers that focus on small tasks — such as experimenting with one particular meta-block header component.

To conclude the library introduction, these are a few examples of its intended use cases:

- Deserialize a compressed file, and analyze parts of its structure
- Deserialize a compressed file, apply a transformer, and serialize the result into a file
- Read an uncompressed file, apply an encode-transform pipeline, and serialize it into a file

4.1 Object Representation of a Brotli Structure

This section gives an overview of the data in `BrotliFileStructure` and component classes. The data in component classes is immutable.

Brotli File Structure	**Brotli File Parameters**	**(Abstract) Meta-Block**
Brotli File Parameters	Sliding Window Size	Uncompressed Data Length
List of Meta-Blocks	Dictionary[18]	

Figure 10: Overview of a Brotli compressed file structure.

Meta-blocks are divided into 4 sub-types inheriting from the abstract meta-block type:

- **Last-Empty** marks a last & empty meta-block. Brotli forbids uncompressed meta-blocks from also being last, appending an empty meta-block to the end of the file avoids that.
- **Padded-Empty** meta-block has no output data, but it skips part of the bit stream. The class exposes any data hidden inside the skipped chunk as an array of bytes.
- **Uncompressed** meta-block contains an array of bytes that represent uncompressed data.
- **Compressed** meta-block contains a header and data section that must be decompressed.

4.1.1 Meta-Block Header

The header fields are ordered by their appearance in the bit stream. The × sign indicates that the component appears once for each listed category mentioned in section 3.7.

Compressed Meta-Block Header
Block Type Info × `[L,I,D]`
Distance Parameters
Literal Context Mode Array
Context Map × `[L,D]`
Huffman Tree Array × `[L,I,D]`

Block Type Info	**Distance Parameters**	**Context Map**
Block Type Count	Postfix Bits	Tree Count
Initial Length	Direct Code Bits	Context Map Array
Huffman Tree for Block Types		
Huffman Tree for Lengths		

Figure 11: Object representation of a compressed meta-block header.

[18] The default Brotli dictionary is embedded in the library, but the library supports custom dictionaries as well.

4.1.2 Meta-Block Data

The data section consists of insert&copy and block-switch commands. Although block-switch commands are incorporated into insert&copy commands, the implementation stores them as separate lists in the meta-block data component. During deserialization/serialization, the block-switch command lists act as queues, so that the currently processed insert&copy command can pull the next block-switch command out of the queue as needed.

Compressed Meta-Block Data
List of Insert&Copy Commands
List of Block-Switch Commands × `[L,I,D]`

Insert&Copy Command	**Block-Switch Command**
List of Literals[19]	Block Type
Copy Length	Block Length
Distance[20]	

Figure 12: Object representations of compressed meta-block data section.

Commands only store decoded data, thus discarding information about which code — length code, distance code, block type code, block length code — was used to encode them. We will come back to that in regards to serialization in section 4.4.

4.2 Deserialization

All component classes define a deserialization procedure that takes a `BitReader` and produces an instance of the component. Larger components call into the deserialization procedures of smaller components. Deserialization follows the decoding algorithm defined by the Brotli specification[2].

Some components require additional context — for example, an insert&copy command needs access to its meta-block's header, and a `BrotliGlobalState` object. The context type is a generic type parameter of the procedure, the concrete context object is provided by the caller.

A `BitReader` acts as a cursor within a `BitStream`, keeping track of the current position and exposing an interface to read the next bit or group of bits. An extension of the `BitReader` marks the read bits in a nested hierarchy of text labels, which can then be displayed and navigated in the GUI application, or saved into a text file in the CLI application. A plain text representation of the entire structure allows using external text processing tools, such as `diff` or `grep`, for comparison and analysis.

[19] Insert length is implicit, based on the number of literals in the list.

[20] Valid distances cannot be negative, so negative values are used to represent special cases (commands with no *copy* part, and distance code 0 which is treated specially).

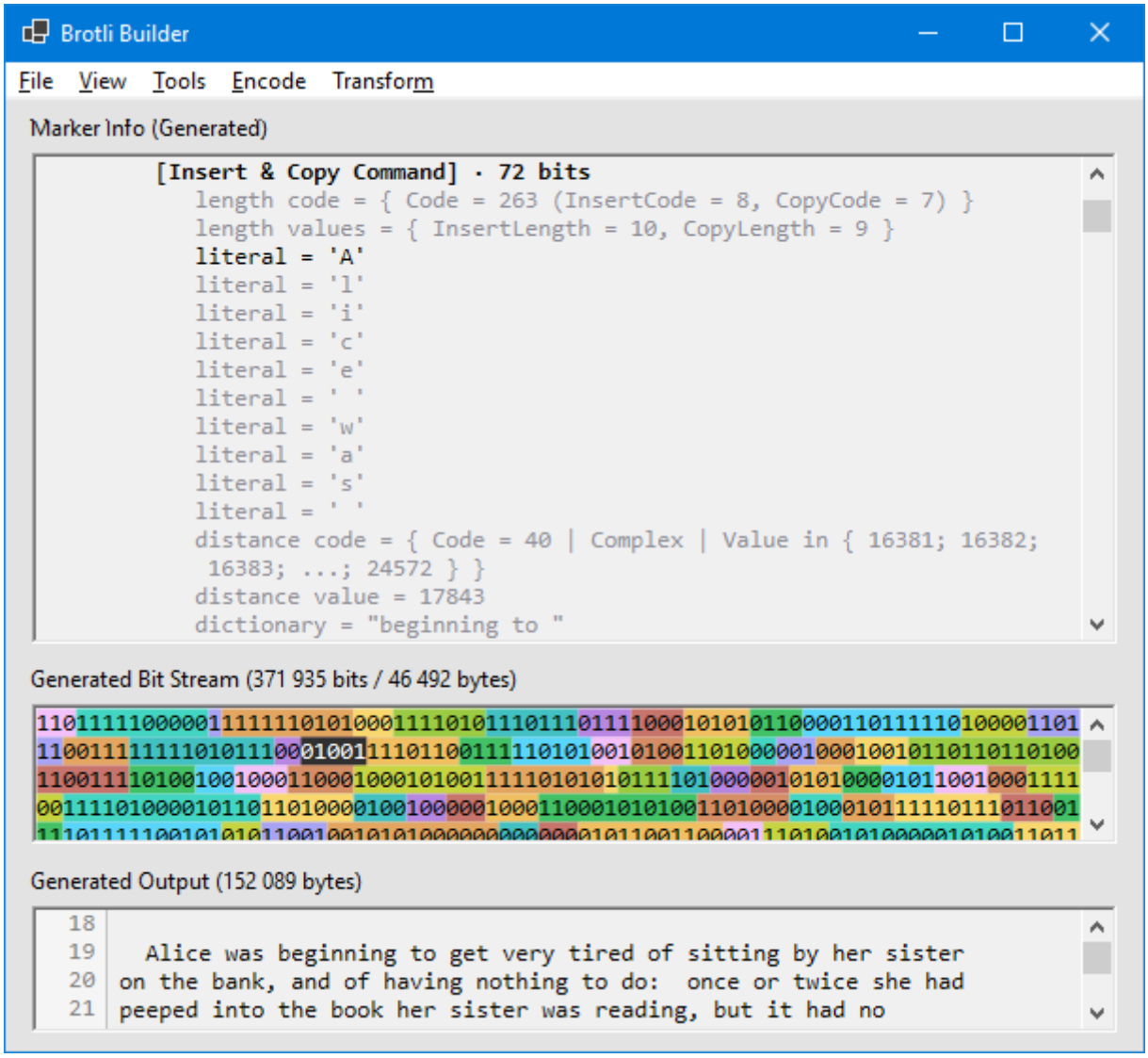


Figure 13: Example of marked bits in the GUI application. The image shows a sequence of bit highlighted in the `Bit Stream` panel, and its label highlighted in the `Marker Info` panel. The bit sequence encodes the literal 'A', as part of an insert&copy command that produces the text "Alice␣was␣beginning␣to␣".

4.3 Decompression

The standard decompression algorithm produces uncompressed data as it reads and decodes each meta-block. In the custom implementation, we generate an object representation of each meta-block in which the data is already decoded, therefore we only have to extract it from the meta-blocks.

All four meta-block types define a `Decompress` method that takes a `BrotliGlobalState` object, which — as mentioned at the beginning of section 4 — handles all state and output. Its

default output processor only keeps uncompressed data the size of the sliding window, but we can choose to use an output processor that keeps all uncompressed data.

To decompress a `BrotliFileStructure`, initialize a `BrotliGlobalState` with the custom output processor. Then for each meta-block, call the `Decompress` method which works as follows:

- **Empty** meta-blocks output nothing
- **Uncompressed** meta-blocks output their uncompressed bytes
- **Compressed** meta-blocks do the following for each insert&copy command:
  1. Output all literals
  2. If the command has a *copy* part, determine its type based on the current state:
     - A backward reference outputs the byte at position given by *distance* in a loop repeated *copy length* times
     - A dictionary reference unpacks the word index and transform ID from *distance*, reads the word using *copy length* and word index, transforms it, and outputs it

4.4 Serialization

Serialization is defined similarly to deserialization. All component classes define a serialization procedure, which encodes each component into a `BitWriter` — the counterpart of `BitReader` — often also requiring a context object. Some components depend on an additional parameter object of type `BrotliSerializationParameters`.

The need for a special parameter object comes from the fact that several components of the Brotli format can be encoded into the bit stream in multiple valid ways. Finding the most optimal encoding is often impractical due to the sheer amount of combinations. Instead, developers come up with heuristics that find a good-enough solution to these kinds of problems. The parameter object is a set of user-provided heuristics (`C#` function delegates) that make decisions about how to encode certain components.

As the Brotli specification only explains the decoding process, all serialization procedures were reverse-engineered[21] from descriptions of the format & decoding algorithm. Naturally, this resulted in differences between files created by the official compressor, and the same files after they were deserialized and serialized again using the library.

The following sections will explore components where these differences emerged, explain their bit representation and serialization procedure, and compare the custom and official implementation on the test corpus. We will skip components whose bit representation is unambiguous, as their serialization process usually involves simple calculations and/or lookup tables.[22]

[21] Some heuristics from the official compressor were adapted into the library afterwards and are present in the current version, partly to allow for experimentation with parts of the official implementation, partly to retain *some* consistency with the official implementation as the default behavior.

[22] The official compressor employs many optimizations and arcane-looking computations for performance reasons. The custom implementation's source code favors simplicity over performance.

An important point is that serialization needs a fully defined `BrotliFileStructure` object,[23] where each meta-block header has all the necessary information to encode that meta-block's data. It is not allowed to modify any components.[24] If, for example, an insert&copy command encoded the literals `abc` but 'c' was missing in the relevant Huffman tree, the serialization procedure would throw an exception — the only way to resolve the issue would have involved modifying the Huffman tree to add the missing symbol, which is prohibited. Section 4.5 will explore ways of (re)generating headers for new or modified meta-blocks.

4.4.1 Serializing Insert&Copy Commands

Let us repeat what parts an insert&copy command is made of, and add a few previously omitted details. An insert&copy command begins with a **length code**, which encodes three parameters:

- Insert length (using an intermediary *insert code*)
- Copy length (using an intermediary *copy code*)
- Whether the *implicit distance code zero* (`IDCZ`) mark is present

The length code is immediately followed by an amount of **literals** equal to the insert length. Afterwards, one of the following situations happens:

- If the meta-block expects no more output, the command has no *copy* part and the meta-block is terminated.
- If the `IDCZ` mark is present, the command uses **distance code zero** — which repeats the previous distance — without reading anything from the bit stream.
- Otherwise, the command ends with an explicit **distance code** in the bit stream, which encodes the concrete distance.

The **length codes**, **distance codes**, and **literals** are the 3 main categories of symbols represented by Huffman trees. In the bit stream, any of these symbols may be preceded by a block-switch command.

As mentioned previously, commands only keep the decoded data — they do not remember which codes were used to encode that data. To serialize the 3 main categories of symbols, we proceed as follows:

- When writing the lengths, the Huffman tree is searched breadth-first for a length code that can encode the lengths and the `IDCZ` mark. Only one such code can exist.
- When writing a literal, the Huffman tree is searched using a fast lookup structure, which directly maps literals to their paths in the tree.

[23] The streaming APIs do not need all meta-blocks at once, but they do need `BrotliFileParameters`, `BrotliGlobalState`, and a fully defined meta-block object.

[24] The library guarantees that for any valid `BrotliFileStructure` or individual component class, serializing and then deserializing it yields an identical copy of it.

- When writing the distance, we must remember that (1) some codes refer to previously seen distances, and (2) some consume a known amount of additional bits from the bit stream. Consequently, we have to consider two factors:

  1. Multiple codes may be able to represent the same value.
  2. A code, which has the shortest path in the tree, may not actually be the most efficient if it requires additional bits.

  To find the shortest distance code, the Huffman tree is searched for all codes that can encode the concrete distance, and the one for which (*path length* + *additional bit count*) is the smallest is chosen.

The laid out logic always finds the most efficient command encoding for a given meta-block header. In contrast, the official compressor immediately assigns length codes and distance codes as it generates the commands. It also tweaks path lengths of symbols in Huffman trees to take better advantage of special run-length codes that will explained in section 4.4.4, but at the expense of making some symbols take more bits. As a result, the official compressor does not always pick the most compact distance code, whereas the custom implementation sees the whole meta-block at once, allowing it to make better decisions when picking distance codes.

4.4.1.1 Distance Code Picking Comparison

The 169 test files compressed using 12 quality settings contained 200 886 450 distance codes. The custom implementation chose a shorter distance code in 152 582 (≈ 0.076%) cases, in total saving 238 986 bits (≈ `29.17 KiB` or 0.0032% of the compressed corpus). Figure 14 shows absolute savings by quality level.

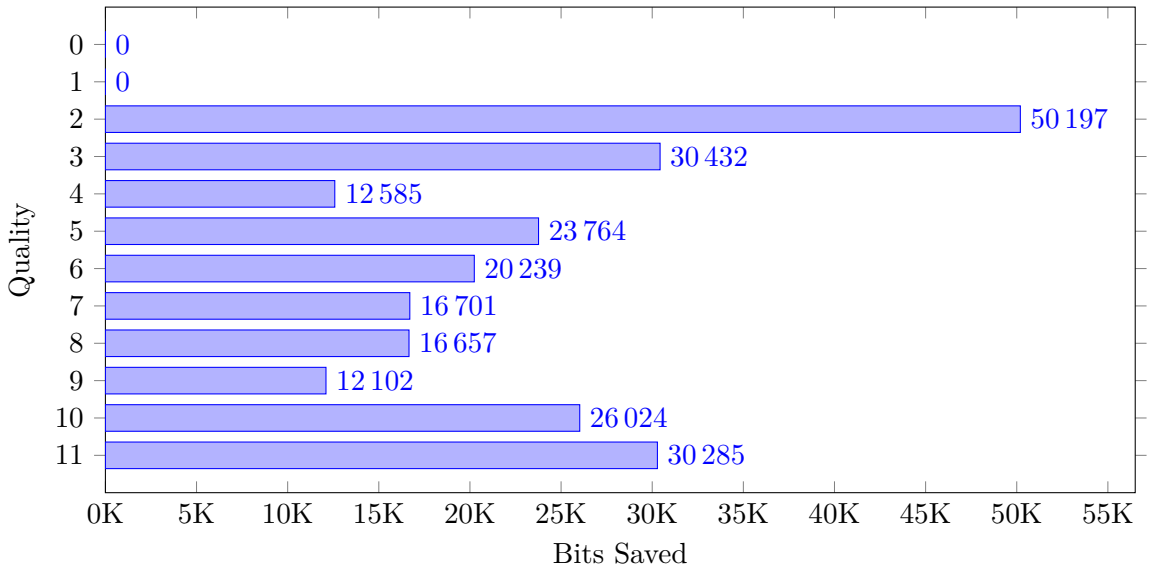


Figure 14: Savings from custom implementation's more efficient distance code picking.

When searching for codes that can encode a certain distance, the final list will always contain exactly 1 **direct** or **complex** code, and 0–4 **last** codes.

The reason for no savings in quality levels 0 and 1 is that out of the 16 possible **last** codes, these quality levels only use code zero which repeats the last distance as-is. We will talk about code zero in section 4.5.2, but for now it is important to know that the decision to use (or not use) code zero in a command is set in stone, and cannot be changed during serialization.

Although this shows a potential for improvements in the official compressor, the savings are minuscule and likely not worth the development attention that could be used elsewhere.

4.4.2 Serializing Block-Switch Commands

A block-switch command sets the current block type and new counter value for one of the 3 categories of symbols in insert&copy commands. The block type and length are determined using a block type code and block length code respectively. Both codes are stored using Huffman trees, but block length codes are unambiguous so this section will not pay attention to them.

Each category can define up to 256 distinct block types per meta-block. The Huffman tree alphabet for type codes has 258 symbols — the first 2 symbols refer to previously seen block types, the remaining 256 symbols map directly to the 256 possible block types.

Block type codes do not require any additional bits unlike distance codes, so a breadth-first search of the Huffman tree is sufficient to find the most compact one. In contrast, the official compressor tests block type codes in a fixed order at the time it generates the command.

4.4.2.1 Block Type Code Picking Comparison

The 169 test files compressed using 8 highest quality settings — those which perform block splitting — contained 303 725 block-switch commands. The custom implementation chose a shorter block type code in 5 560 (≈ 1.83%) cases, in total saving 8 507 bits (≈ `1.04 KiB` or 0.00012% of the compressed corpus). Figure 15 shows absolute savings per quality level:

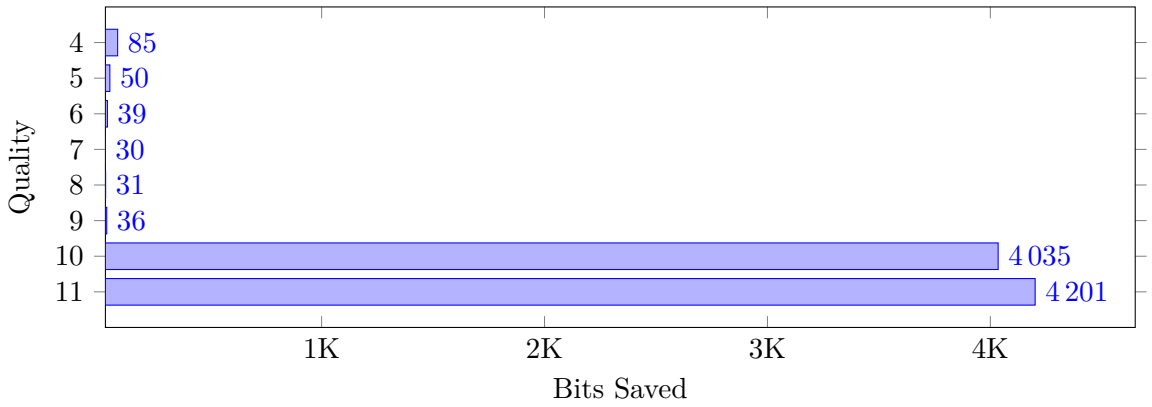


Figure 15: Savings from custom implementation's more efficient block type code picking.

4.4.3 Serializing Context Maps

A context map maps every possible $\langle$ *block type*, *context ID* $\rangle$ pair to an index in an array of Huffman trees. Each meta-block header defines one context map for literals (64 context IDs) and one context map for distance codes (4 context IDs). The mapping is implemented as a single byte array of size (*total block types* $\times$ *total context IDs*).

In the bit stream, a context map begins with a variable-length code that reads a value between 1–256. This value indicates the amount of Huffman trees, which also tells us the size of the Huffman tree array, and the range of indices that can appear in the context map data.

If the amount of Huffman trees equals 1, the context map is *trivial* — all indices are zero. Otherwise, we have to reconstruct the entire byte array, which may span up to 256×64 elements or `16 KiB` for literals, and 256×4 elements or `1 KiB` for distance codes. Storing the entire array as a contiguous sequence of bytes would be highly inefficient (see figure 18 in the next section). Predictably, Brotli employs several techniques to greatly compact them.

Firstly, we can encode the array of bytes using a Huffman tree with symbols 0–255, denoting their respective byte values. This also lets us omit unused symbols, yielding significant savings in meta-blocks with small amounts of Huffman trees.

Secondly, presume that context maps will have consecutive occurrences of the same value, also called *runs*. For example, we could assign a separate Huffman tree to each block type. Figure 16 shows such context map for literals with 3 block types, resulting in 3 runs of 64 values each:

$$\underbrace{0, \ldots, 0}_{64\times}, \underbrace{1, \ldots, 1}_{64\times}, \underbrace{2, \ldots, 2}_{64\times}$$

Figure 16: Example context map with long runs.

Brotli heavily optimizes these kinds of context maps by combining two general compression techniques:

1. **Move-to-front transform** takes a set alphabet, but instead of encoding the symbols, it encodes each symbol's ordinal in the alphabet — then, that symbol is moved to the first position (front) in the alphabet[5]. See table 2 for an example of applying the move-to-front transform to an English word, and pay attention to low numbers where letters repeat.[25]

   During serialization, Brotli applies the move-to-front transform on the context map byte array, whose alphabet consists of the byte values 0–255. During deserialization, it uses an inverse transform described in the Brotli specification to retrieve the original values. Although Brotli makes the transform optional, the official compressor enables it in all context maps.

[25] One possible way to exploit this would be to use Huffman coding, assigning short bit sequences to low numbers and long bit sequences to high numbers.

Table 2: Example of applying the move-to-front transform in individual steps. The alphabet consists of English letters `a-z` where `a = 1` and `z = 26`.

Sequence	Original	Transformed	Next Alphabet State
-	-	-	abcdefghijklmnopqrst...
m	13	13	mabcdefghijklnopqrst...
mo	13, 15	13, 15	omabcdefghijklnpqrst...
mon	13, 15, 14	13, 15, 15	nomabcdefghijklpqrst...
mons	13, 15, 14, 19	13, 15, 15, 19	snomabcdefghijklpqrt...
monso	13, 15, 14, 19, 15	13, 15, 15, 19, 3	osnmabcdefghijklpqrt...
monsoo	13, 15, 14, 19, 15, 15	13, 15, 15, 19, 3, 1	osnmabcdefghijklpqrt...
monsoon	13, 15, 14, 19, 15, 15, 14	13, 15, 15, 19, 3, 1, 3	nosmabcdefghijklpqrt...
monsoons	13, 15, 14, 19, 15, 15, 14, 19	13, 15, 15, 19, 3, 1, 3, 3	snomabcdefghijklpqrt...

Figure 17 is an example of a small context map for distances, which uses a separate Huffman tree for each of its 3 block types. The figure shows its values before and after applying the move-to-front transform, turning it from a sequence where all values have equal probability into a sequence where 0 is the most frequent.

$$0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2$$
$$\downarrow$$
$$0, 0, 0, 0, 1, 0, 0, 0, 2, 0, 0, 0$$

Figure 17: Example of applying the move-to-front transform to a context map.

2. **Run-length encoding**[5] replaces consecutive occurrences of the same symbol — runs — with an instruction that encodes only one instance of the symbol, and the amount of its occurrences — run length.

   In context maps, Brotli uses run-length encoding for runs of the symbol 0. Non-trivial context maps have a parameter which adds up to 16 intermediary codes into the alphabet of byte values, potentially expanding the alphabet from 256 to 272 symbols. As is usual with intermediary codes, each code can represent a range of run lengths, and consumes additional bits to find the offset within that range.

   If all 16 codes are used, the maximum representable run length is $2^{17} - 1 = 131\,071$. However, 16 383 is the longest run actually possible in Brotli,[26] which matches the maximum run length of code 13 ($2^{14} - 1$).

   In practice, most context maps generated by the official compressor are either similar to the one in figure 17, or they are generated from statistics about the input file and those do not normally end up with very long runs.

[26] A context map for literals with 256 block types would contain $256 \times 64 = 16\,384$ values. The longest run is one less because at least one value must differ — otherwise it would be treated as a trivial context map.

4.4.3.1 Optimization Analysis

Let us digress for a moment to test the usefulness of Huffman coding, move-to-front transform, and run-length encoding in context maps across the test corpus.

From all compressed files, information about every non-trivial context map was compiled into a single list. Figure 18 shows a ratio for each quality level. The ratio was calculated as the sum of all actually used bits, divided by the amount of bits all byte arrays would use if encoded raw.[27] The increasing ratio — decreasing effectiveness — correlates with increasing complexity of context maps in high quality levels.

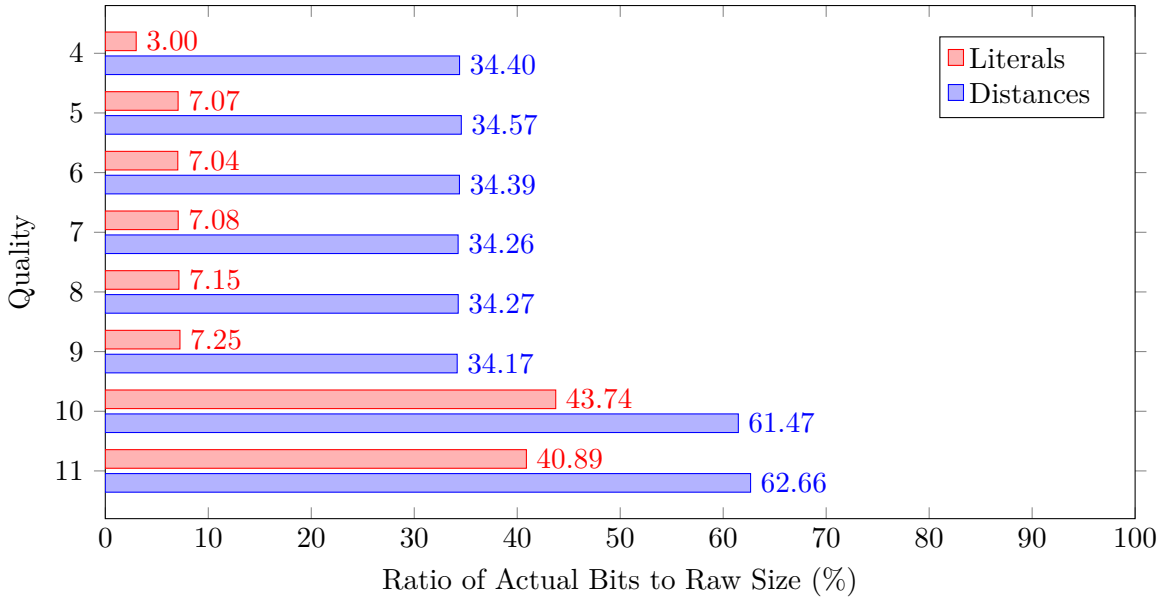


Figure 18: Analysis of context map optimization effectiveness across test corpus files.

4.4.3.2 Serialization Parameters

When tasked with serializing a context map, we have a few decisions to make:

- Do we apply the move-to-front transform?
- What is the best way of encoding each run? Should we use special codes at all?
- What is the best way of encoding the Huffman tree for byte values and special codes?

Let us consider two tricky scenarios.

1. The sequence $(0, 0, 0, 1)$ has a run replaceable by a special code, reducing its size by 1. However, enabling special codes for the context map uses 4 extra bits, and expands the Huffman tree alphabet which in this case adds 2 bits. Our net "savings" are -5 bits.

[27] The actually used bits include the Huffman tree and other metadata needed to decode the context map. If context maps were encoded as raw byte arrays, they would not need this metadata, thus the comparison is fair.

2. A run of 8 zeros is at a boundary — the run is completely covered by special code 3, but can also be encoded using special code 2 followed by a single 0. Which option is better depends on their paths in the Huffman tree, and the encoding of the Huffman tree itself.

All decisions are ultimately passed onto the library user with `BrotliSerializationParameters`. Context maps serialization defines three parameters — function delegates. The move-to-front parameter takes a context map, and returns a boolean that says whether to use the transform. The run-length parameter is given all runs found in the byte array, and a system to accept/reject/split each one. The Huffman tree parameter takes a list of symbol frequencies and returns the tree structure.

The defaults are set to always use move-to-front, replace every run with a special code that fully covers it, and use the classic Huffman coding algorithm to generate the tree.

4.4.3.3 Comparison

This comparison tests combinations of strategies on the test corpus. It is important to acknowledge the limits of this exercise — we are evaluating strategies with fixed rules, and only on context maps generated by the official compressor. The best overall strategy we find will certainly not be the best for all possible context maps. Huffman trees are also a factor — their serialization will be discussed in section 4.4.4, but they do mildly differ from the official compressor.

The data was obtained by individually serializing all non-trivial context maps in the test corpus and counting the bits. Figures 19 and 20 show the total amounts of bits for each quality level. Tables 3 and 4 note which strategies encoded context maps the best, and which were tied with another strategy. All combinations of the following parameters were tested:

- **MTF** (move-to-front transform)
  - **Yes** enables the transform
  - **No** disables the transform
- **RLE** (run-length encoding)
  - **No** does not use special codes
  - **Full** encodes all runs using special codes
  - **Split** encodes all runs using special codes, but if shortening the run by 1 also shortens the code, it encodes the shorter run followed by a plain 0

The results indicate that using the move-to-front transform is in almost all cases the best option, but it does lose some effectiveness in the two highest quality levels that generate the most complex context maps. There is no single run-length encoding strategy that works best in all quality levels and types of context maps, but choosing either **Full** or **Split** strategy yields sufficiently good results.

Table 3: % of times a strategy for non-trivial context maps for literals was the best/tied.

Q	MTF = No RLE = No	MTF = No RLE = Full	MTF = No RLE = Split	MTF = Yes RLE = No	MTF = Yes RLE = Full	MTF = Yes RLE = Split
4	0.0% / 0.0%	0.0% / 0.0%	0.0% / 0.0%	0.0% / 0.0%	18.4% / 28.7%	52.9% / 28.7%
5	0.0% / 0.0%	0.0% / 0.0%	0.0% / 0.0%	25.9% / 0.0%	13.6% / 34.6%	25.9% / 34.6%
6	0.0% / 0.0%	0.0% / 0.0%	0.0% / 0.0%	26.3% / 0.0%	12.5% / 35.0%	26.3% / 35.0%
7	0.0% / 0.0%	0.0% / 0.0%	0.0% / 0.0%	26.3% / 0.0%	11.3% / 36.3%	26.3% / 36.3%
8	0.0% / 0.0%	0.0% / 0.0%	0.0% / 0.0%	26.3% / 0.0%	11.3% / 35.0%	27.5% / 35.0%
9	0.0% / 0.0%	0.0% / 0.0%	0.0% / 0.0%	26.9% / 0.0%	14.1% / 34.6%	24.4% / 34.6%
10	0.0% / 0.6%	2.9% / 1.1%	1.1% / 1.1%	69.1% / 1.1%	18.9% / 1.7%	4.0% / 2.3%
11	0.6% / 3.5%	1.7% / 2.9%	0.0% / 2.9%	64.4% / 4.0%	19.6% / 1.7%	5.2% / 2.3%

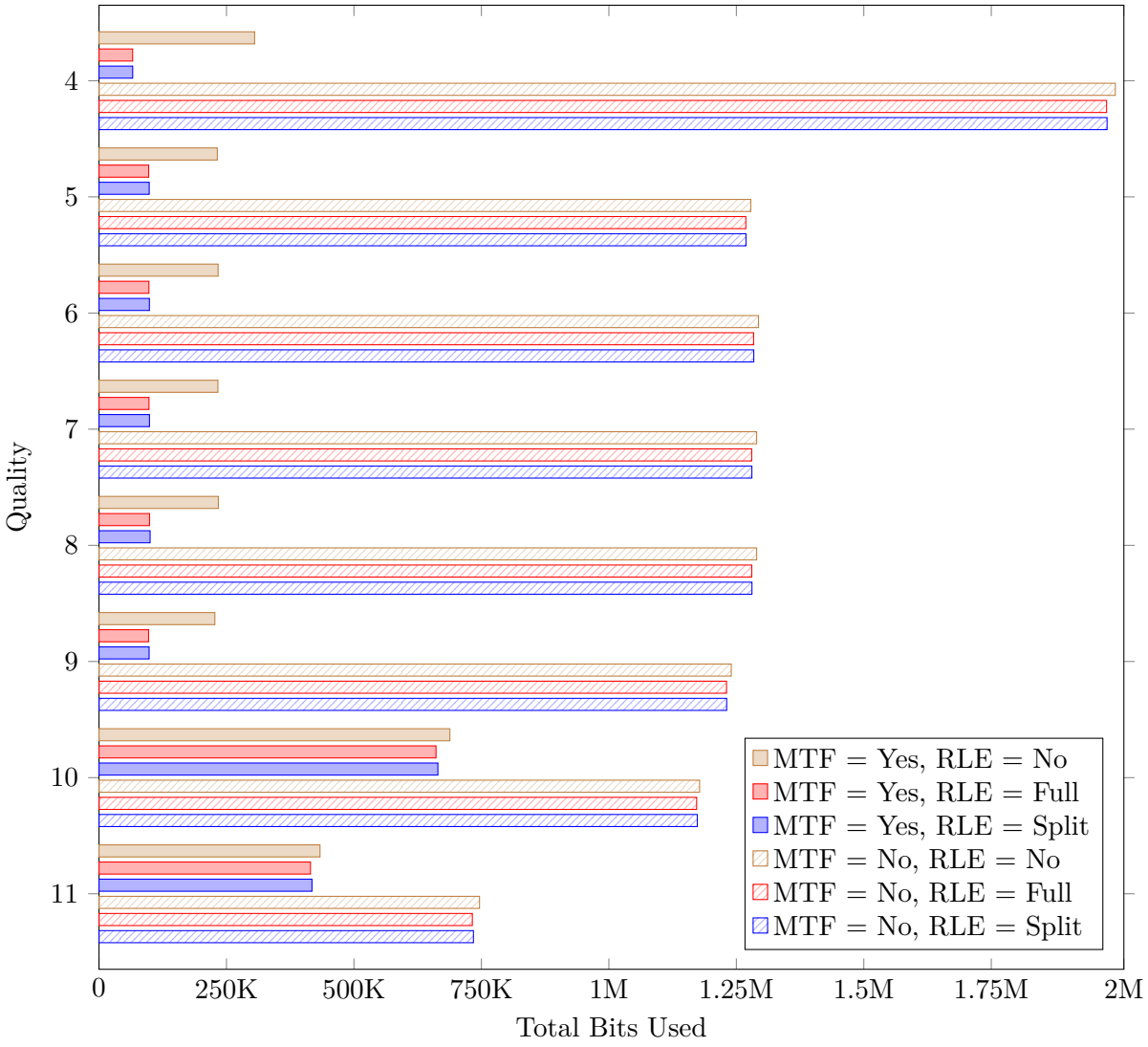


Figure 19: Comparison of serialization strategies in non-trivial context maps for literals.

Table 4: % of times a strategy for non-trivial context maps for distance codes was the best/tied.

Q	MTF = No RLE = No	MTF = No RLE = Full	MTF = No RLE = Split	MTF = Yes RLE = No	MTF = Yes RLE = Full	MTF = Yes RLE = Split
4	0.0% / 18.8%	0.0% / 0.0%	0.0% / 0.0%	47.1% / 19.6%	1.5% / 1.4%	30.4% / 2.2%
5	0.0% / 19.4%	0.0% / 0.0%	0.0% / 0.0%	49.3% / 20.9%	1.5% / 0.0%	28.4% / 1.5%
6	0.0% / 22.1%	0.0% / 0.0%	0.0% / 0.0%	46.3% / 25.0%	0.7% / 1.5%	26.5% / 4.4%
7	0.0% / 21.2%	0.0% / 0.0%	0.0% / 0.0%	41.6% / 21.9%	0.0% / 0.7%	35.8% / 1.5%
8	0.0% / 21.0%	0.0% / 0.0%	0.0% / 0.0%	44.9% / 21.7%	1.5% / 1.4%	30.4% / 2.2%
9	0.0% / 19.9%	0.0% / 0.0%	0.0% / 0.0%	43.4% / 22.1%	0.7% / 0.7%	33.1% / 3.0%
10	5.3% / 62.9%	1.3% / 2.0%	0.0% / 61.6%	6.0% / 63.6%	0.0% / 9.9%	2.0% / 62.3%
11	1.5% / 48.5%	3.0% / 3.8%	0.0% / 43.2%	18.2% / 62.1%	0.8% / 5.3%	0.0% / 58.3%

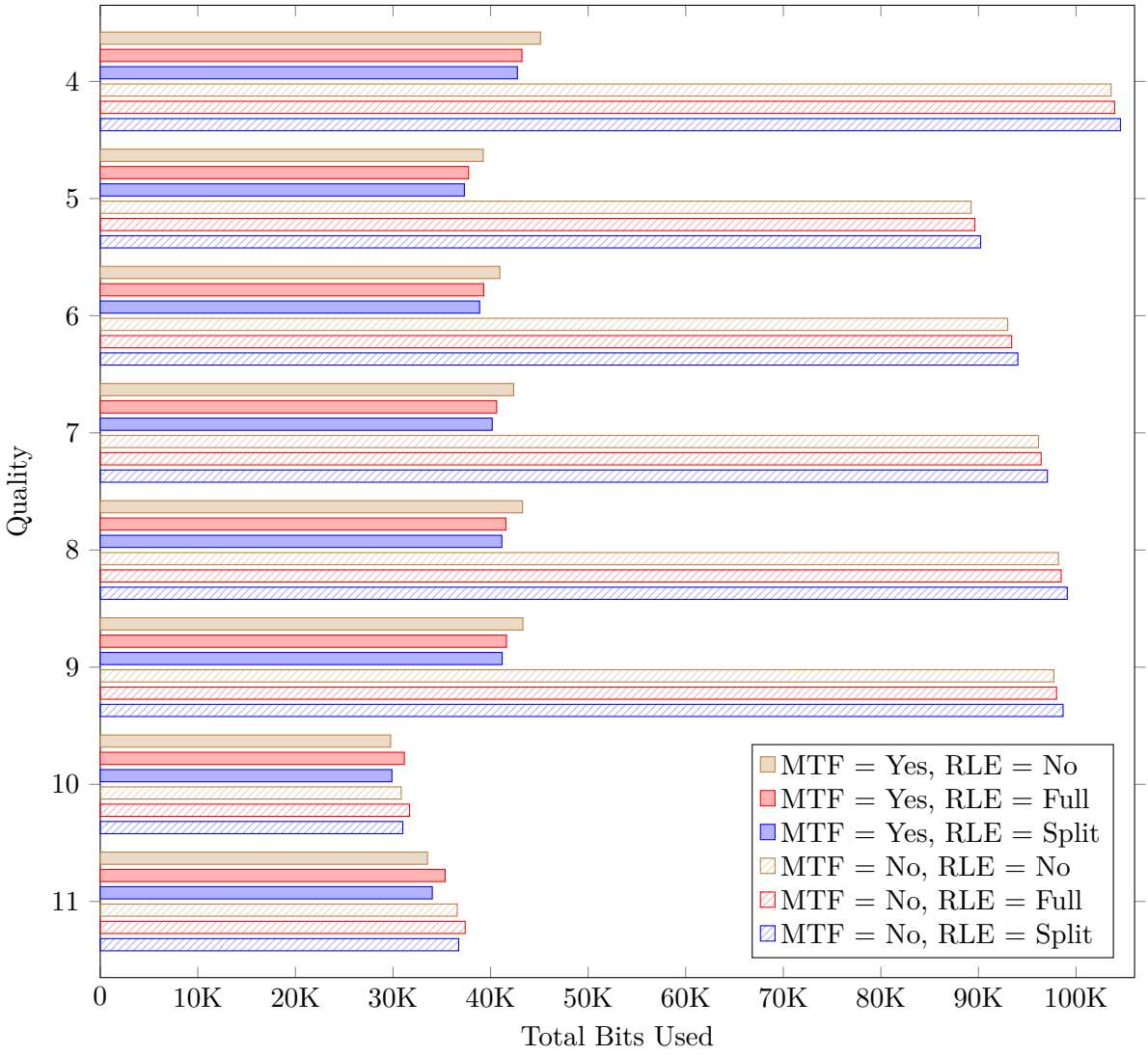


Figure 20: Comparison of serialization strategies in non-trivial context maps for distance codes.

4.4.4 Serializing Huffman Trees

The Brotli format has two ways to encode Huffman trees — **simple** and **complex**. The **simple** encoding handles trees with 1–4 symbols, in five possible shapes displayed in figure 21. A tree with only 1 symbol is a special case, where reading the symbol does not consume any bits from the bit stream.

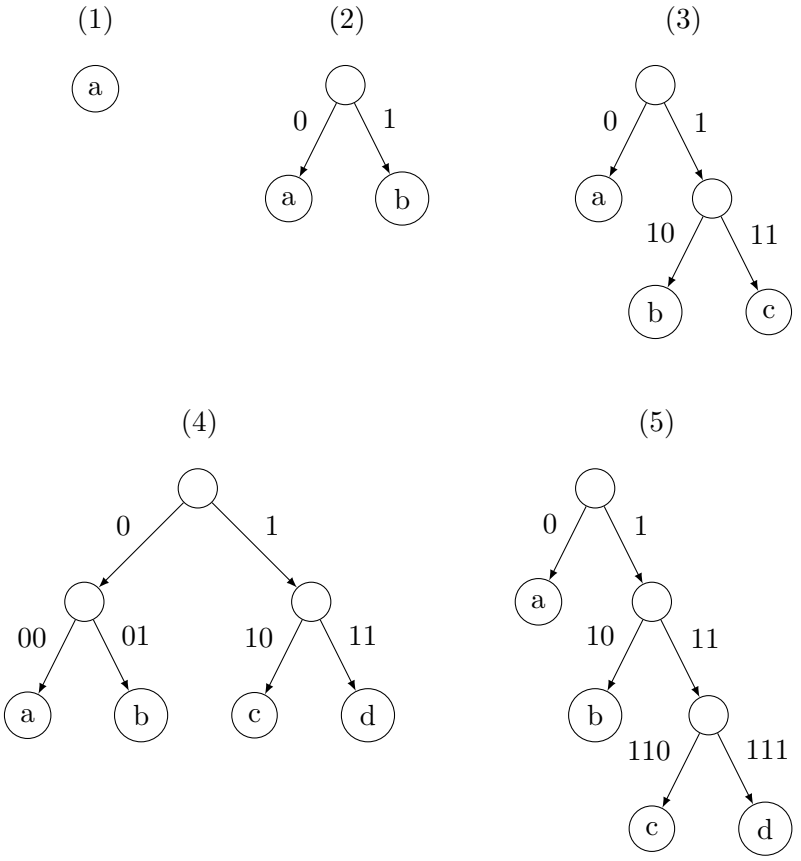


Figure 21: Shapes of Huffman trees that can use the simple form of encoding.

The **complex** form is based on canonical Huffman coding[2], which is also used by other compression standards such as DEFLATE[1].

Canonical Huffman coding only stores the path lengths of each symbol, following the order in which the symbols appear in the alphabet. Brotli does two things to condense them even further. Firstly, it uses special codes as a form of run-length encoding:

- **Code 0** means the current symbol is not present in the tree
- **Codes 1–15** assign the current symbol a path length of 1–15
- **Code 16** repeats the last assigned path length over the next several symbols
- **Code 17** repeats code 0 over the next several symbols, efficiently skipping unused symbols in large alphabets

Secondly, these codes are read with the help of a *secondary* Huffman tree, which is freshly generated for each *primary* tree. The secondary tree is also stored using canonical Huffman coding, and its own path lengths are stored with a variable-length code defined in the Brotli specification.

Codes 16 and 17 are followed by 2 and 3 extra bits respectively, which are used in the computation of the run length. When a repetition code itself repeats, the previously read extra bits act as a multiplier of the current result. This is best shown on an example of code 16's first several run lengths and how they are represented:

16 +00	$= 3 + 0 = 3$	
16 +01	$= 3 + 1 = 4$	
16 +10	$= 3 + 2 = 5$	
16 +11	$= 3 + 3 = 6$	
16 +00 16 +00	$= 3 + (4 \times 1) + 0 = 7$	16 +01 16 +00 $= 3 + (4 \times 2) + 0 = 11$
16 +00 16 +01	$= 3 + (4 \times 1) + 1 = 8$	16 +01 16 +01 $= 3 + (4 \times 2) + 1 = 12$
16 +00 16 +10	$= 3 + (4 \times 1) + 2 = 9$	16 +01 16 +10 $= 3 + (4 \times 2) + 2 = 13$
16 +00 16 +11	$= 3 + (4 \times 1) + 3 = 10$	16 +01 16 +11 $= 3 + (4 \times 2) + 3 = 14$
16 +10 16 +00	$= 3 + (4 \times 3) + 0 = 15$	16 +11 16 +00 $= 3 + (4 \times 4) + 0 = 19$
16 +10 16 +01	$= 3 + (4 \times 3) + 1 = 16$	16 +11 16 +01 $= 3 + (4 \times 4) + 1 = 20$
16 +10 16 +10	$= 3 + (4 \times 3) + 2 = 17$	16 +11 16 +10 $= 3 + (4 \times 4) + 2 = 21$
16 +10 16 +11	$= 3 + (4 \times 3) + 3 = 18$	16 +11 16 +11 $= 3 + (4 \times 4) + 3 = 22$

16 +00 16 +00 16 +00 $= 3 + \Big(4 \times \big((4 \times 1) + 1\big)\Big) + 0 = 23$

16 +00 16 +00 16 +01 $= 3 + \Big(4 \times \big((4 \times 1) + 1\big)\Big) + 1 = 24$

$\vdots$

16 +11 16 +11 16 +11 $= 3 + \Big(4 \times \big((4 \times 4) + 4\big)\Big) + 3 = 86$

$\vdots$

etc.

Figure 22: Computing run lengths from repeated code 16 & extra bits in Huffman tree serialization.

The custom implementation includes the classic Huffman coding algorithm based on symbol frequencies, and also the canonical Huffman coding algorithm. Brotli additionally imposes a

depth limit — primary trees must have path lengths of at most 15, secondary trees at most 5. In the custom implementation, depth-limiting uses a heuristical approach described by Charles Bloom, although solving the problem optimally is possible using the package-merge algorithm[8].

4.4.4.1 Serialization Parameters

The amount and complexity of Huffman trees make them the bulk of meta-block headers — across the test corpus, they make up $\approx 94.4\%$ of all header bits.

`BrotliSerializationParameters` exposes two parameters — function delegates. Similarly to context maps, one parameter controls run-length encoding — this time in terms of codes 16 and 17 — by asking the parameter how to handle each run in the sequence, and one parameter controls generation of the secondary Huffman tree. By default, the library encodes runs using a heuristic adapted from the official compressor:

- If the tree's symbol alphabet contains 50 symbols or fewer, run-length encoding is skipped
- Code 16 is used for runs of non-zero codes if $\sum_{i=1}^{n} length_i > 2n$, for $length_i \geq 4$
- Code 17 is used for runs of zeros if $\sum_{i=1}^{n} length_i > 2n$
- Code 16 runs of length 7 are split[28] into 1 normal code and a run of length 6
- Code 17 runs of length 11 are split[29] into 1 normal code and a run of length 10

Using the same heuristic makes Huffman tree encoding nearly identical to the official compressor. A difference still exists in generation of the secondary tree — one interesting optimization the official compressor applies to some trees is equalizing adjacent symbols' path lengths to form longer runs. This condenses the meta-block header at the expense of using more bits in the data section, as the tree is no longer optimal. Such optimization could be reimplemented using the parameters that control how Huffman trees are generated — that would affect secondary trees mentioned in this section, and trees generated for context map values. In order to apply this optimization to other Huffman trees, such as the ones for insert&copy command elements, we would have to rebuild the meta-block header, which will be discussed in section 4.5.

4.4.5 Concluding Serialization

This section takes a brief look at the overall differences in compressed size between the custom and official implementations. To reiterate an important point, serialization is not allowed to modify data in any of the components, it does however regenerate metadata about how certain components are encoded in the bit stream.

Figure 23 shows the differences after taking every file from the test corpus, deserializing it, and serializing it again with default `BrotliSerializationParameters`. To eliminate outliers,

[28] The last 2 rules are special cases for the first run lengths that require 2 successive repetition codes. The heuristic assumes it is always better to encode one of the symbols normally and reduce the run length by one.

[29] See previous footnote.

files whose uncompressed size was less than `512 B` were excluded (8 files in total). Plot whiskers show the minimum and maximum. The second plot zooms in on quality levels 3–11.

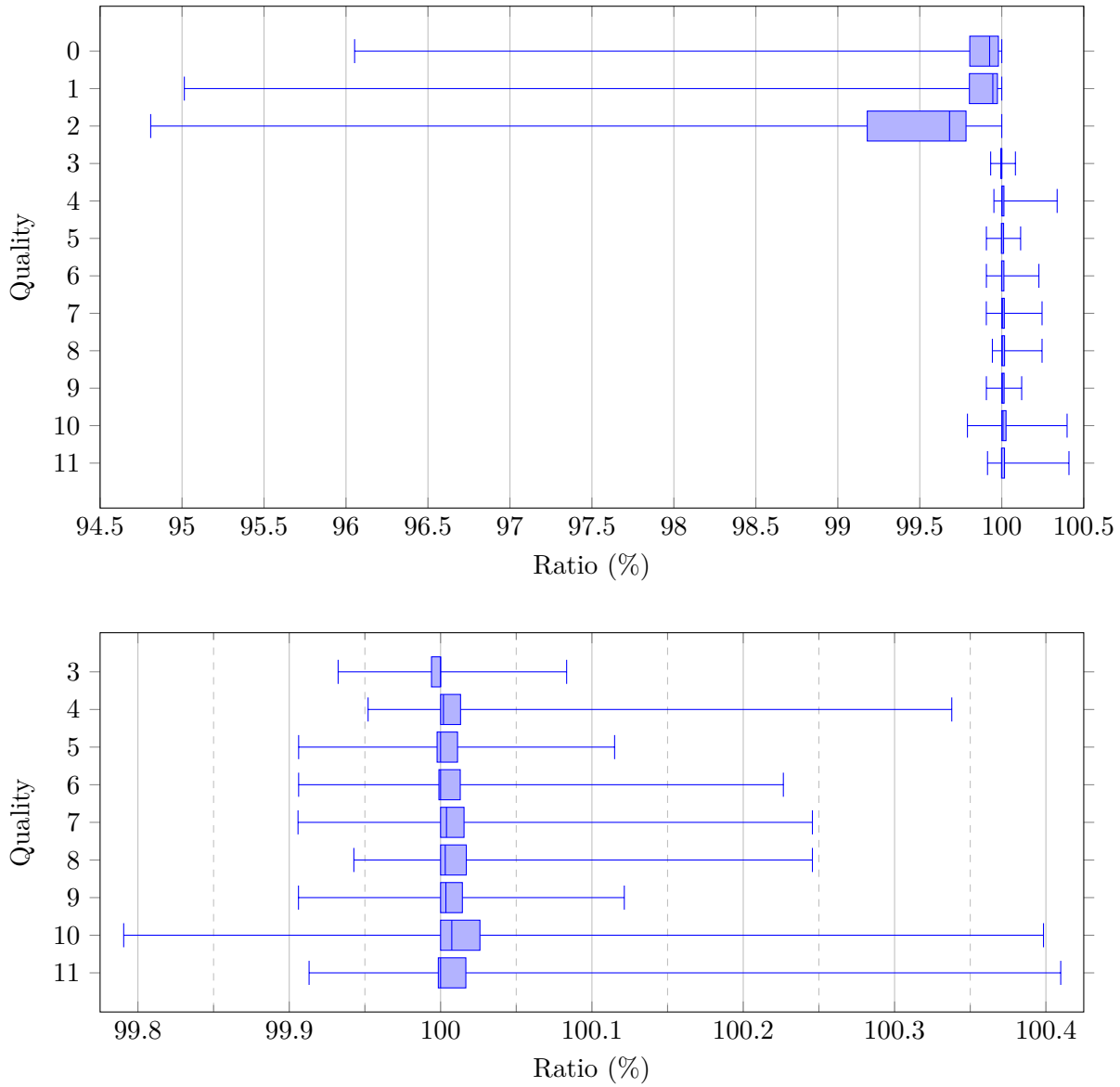


Figure 23: Compressed file size ratios (custom serialization ÷ official implementation).

Largest improvements are found in quality levels 0–2, mainly because the secondary trees they use to encode Huffman trees are predefined, while the custom implementation generates them from scratch. Low quality levels also tend to generate higher quantities of shorter meta-blocks, giving the custom implementation more Huffman trees — opportunities — to optimize.

4.5 (Re)building the Structure

Although serialization is a crucial part of supporting the Brotli format, we want the ability to modify an existing `BrotliFileStructure` or create a completely new one. The library streamlines this process, making it less error-prone than if we tried to do it manually.

This section introduces several builder APIs that automate construction of meta-block header components — such as Huffman trees and codes used in commands — from the data we want to encode. We will go over examples of using the API, and some implementation details of the builders and related utilities.

4.5.1 Dictionary Implementation

A `BrotliDictionary` definition consists of 3 parts:

- **Data source** that can seek and retrieve the raw bytes that form dictionary words. A source can be a file or data stream, or even a simple byte array in memory.
- **Word transforms** that govern how prefixes, suffixes, and transforms are applied to the retrieved words, as described in section 3.4.1.
- **Format definition** that describes the data source's structure, knows how to locate words in the data source by their length and index, and packs/unpacks the word index and transform ID to/from dictionary references.

The default dictionary sets the format and word transforms according to the standard, and its data source is a file embedded into the library.

Note that while the library makes it possible to use custom dictionaries, it would be up to the user to remember which dictionary was used for which files, and ensure their data sources are available to perform decompression. For that reason, and because the upcoming Brotli format extension would require additional changes for proper support, the library's current implementation of custom dictionaries is not particularly useful for real-world use cases.

4.5.1.1 Index Structure

An index structure finds dictionary references that match the beginning of a given sequence of bytes. A dictionary reference takes the form of a $\langle$ *copy length*, *packed value* $\rangle$ pair.

Internally, the index uses PATRICIA trees. This type of tree was chosen because it has the interface of prefix trees — unlike for example a hash table — and it is much more compact in memory than a classic *trie*. A prefix tree interface matches on a common prefix. We can take an arbitrarily long sequence of bytes, traverse down the tree and at each node add matching entries to a list, and end up with a collection of all dictionary references matching the beginning of the sequence.

For example, let us search the phrase "Back to the future" for matches at least 4 bytes long. Figure 24 shows gradual matches for all prefixes of the phrase, assuming a scenario in which

the entire dictionary — including all transformed words — is stored in a single tree. The end of each row lists matching $\langle\, word\ length\ group,\ word\ index,\ transform\ ID \,\rangle$ triples.

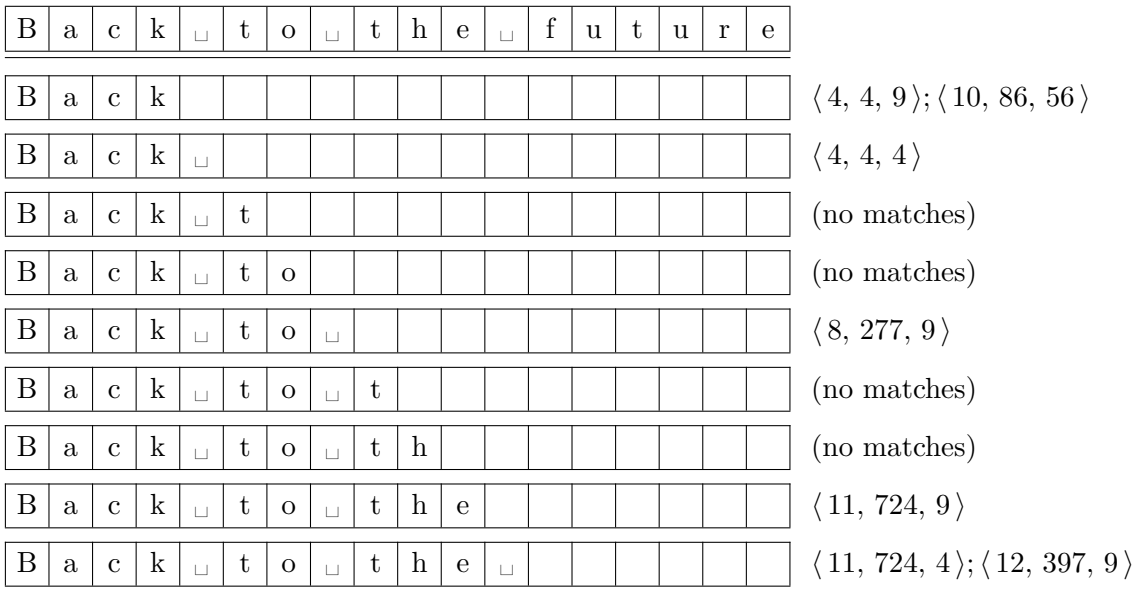


Figure 24: Results of dictionary index lookup on prefixes of the input of length ≥ 4.

We can look at some of the results to find out how they work:

- $\langle\, 4,\ 4,\ 9 \,\rangle$: Word 4 of length 4 is `back`, transform 9 makes it `Back`
- $\langle\, 4,\ 4,\ 4 \,\rangle$: Word 4 of length 4 is `back`, transform 4 makes it `Back␣`
- $\langle\, 10,\ 86,\ 56 \,\rangle$: Word 86 of length 10 is `Background`, transform 56 makes it `Back`
- $\langle\, 12,\ 397,\ 9 \,\rangle$: Word 397 of length 12 is `back␣to␣the␣`, transform 9 makes it `Back␣to␣the␣`

All findings are made available to the caller, so that they can decide which one to pick — longest matches are not always advantageous, as increasing word lengths and especially large transform IDs can greatly inflate distances in insert&copy commands, needing more bits to encode.

A naïve way to construct the index may iterate through all 121 word transforms, apply each to all words in the dictionary, and store everything in a single tree. Although the dictionary index is constructed lazily at first request, even that first request should be reasonably fast, which requires a smarter approach.

Instead, there is a separate PATRICIA tree for each of the 21 preset functions — recall those are **Identity**, **Omit First 1–9**, **Omit Last 1–9**, **Ferment First**, and **Ferment All**. The functions are one by one applied to each word in the dictionary, then words with at least 1 byte remaining are stored in the corresponding tree. Notice that prefixes and suffixes are not included

in the trees, so they must be accounted for during lookup instead. The lookup procedure works as follows:

1. Consider only transforms which have no prefix, or whose prefix matches the beginning of the searched input.
2. For each transform, strip its prefix from the input, then perform lookup in the tree assigned to the transform's function.[30]
3. For each match, strip the transformed word from the input, and ensure the rest matches the transform's suffix.
4. Return all valid matches.

A match comprises the dictionary reference $\langle$ *copy length*, *packed value* $\rangle$ as well as the actual output length to help us select the best result. To construct an insert&copy command, we must know the sliding window size, and the amount of bytes output at the exact point this command will be read. If we are adding literals into the command, they must also be counted. The final distance is: $\mathit{packed\ value} + 1 + \min(\mathit{window\ size},\ \mathit{output\ size} + \mathit{amount\ of\ literals})$.

4.5.2 Building Insert&Copy Commands

`CompressedMetaBlockBuilder` is a builder API for creating and editing meta-blocks. It takes an initial `BrotliGlobalState` that describes the state at the beginning of this meta-block, and optionally an existing meta-block from which it copies the header and data. Figure 25 is a reminder about all components of a compressed meta-block.

Header	**Data**
Block Type Info × `[L,I,D]`	List of Insert&Copy Commands
Distance Parameters	List of Block-Switch Commands × `[L,I,D]`
Literal Context Mode Array	
Context Map × `[L,D]`	
Huffman Tree Array × `[L,I,D]`	

Figure 25: Compressed meta-block header and data components.

Building block-switch commands and information about block types and lengths will be covered in section 4.5.3, and context maps in section 4.5.4. Distance parameters and literal context modes are exposed as read/write properties. The job of a `CompressedMetaBlockBuilder` is to process insert&copy commands, generate Huffman tree arrays based on the commands and other header parameters, and combine everything into the final meta-block object.

Listing 1 is a `C#` code snippet that creates a meta-block from scratch using the builder API.

[30] Many transforms use the same function, so the tree lookup can be memoized.

```
var state = new BrotliGlobalState(BrotliFileParameters.Default);
var builder = new CompressedMetaBlockBuilder(state);

// Generates command 1 that outputs "This␣is␣" from literals "This␣",
// and a copy part that produces "is␣" by copying the last 3 literals.
builder.AddInsertCopy(
    Literal.FromString("This ", UTF8),
    copyLength: 3,
    copyDistance: 3);

// Generates command 2 with literals "test" and no copy part.
builder.AddInsert(Literal.FromString("test", UTF8));

// Every command must either have a copy part, or be the last in a meta-block.
// Subsequent commands will be automatically merged until either
// the meta-block is built, or a command introduces a copy part.

// Merges literals "ing" into command 2.
// The command now produces the text "testing".
builder.AddInsert(Literal.FromString("ing", UTF8));

// The previous command is still missing a copy part.
// This finds "␣data" in the dictionary, encodes it into a copy part,
// and merges with command 2 which will now output "testing␣data".
var dictionary = BrotliFileParameters.Default.Dictionary.Index;
var results = dictionary.Find(UTF8.GetBytes(" data"), minLength: 5);
builder.AddCopy(results.First());

// The previous command now has a copy part, so this
// generates command 3 with literals "!!".
builder.AddInsert(Literal.FromString("!!", UTF8));

// We can check the output size before building.
Assert(builder.OutputSize == "This is testing data!!".Length);

// Build the meta-block and obtain the state for building the next meta-block.
var (metaBlock,nextState) = builder.Build(BrotliCompressionParameters.Default);
```

Listing 1: `CompressedMetaBlockBuilder` API usage examples.

As commands are added, the builder tracks the transitional state so that it (1) knows how many bytes were output so far, (2) remembers the previous distance to substitute matching distances with distance code 0, and (3) calculates distances for dictionary references.

Distance code 0 is treated specially — it does not update the buffer of previously used distances, and it can sometimes be encoded directly in the command length code (the `IDCZ` mark). The library handles special cases concerning distances with an enumeration type `DistanceInfo`, whose instances are either positive integers denoting concrete distances, or one of 3 special values:

- `EndsAfterLiterals`
- `ImplicitCodeZero`
- `ExplicitCodeZero`

This is why decisions about using code 0 are made during command creation. Builder methods that take concrete distances have an optional parameter which, if the passed distance is the same as the previous distance, decides whether to use an implicit code zero, explicit code zero, or encode the distance without code zero. The default behavior is to use implicit code zero if possible, explicit code zero otherwise.

Once all commands and customizable header components are set, the build process can begin:

1. Build block-switch commands (section 4.5.3)
2. Determine sizes of the Huffman tree arrays:
    - For length codes, the size equals the amount of block types for length codes
    - For literals and distance codes, sizes are set in context maps
3. Initialize a symbol frequency counter for each Huffman tree index
4. Simulate all insert&copy and block-switch commands:
    - Determine next block type (= tree index) for length codes
    - Generate the length code and add it to the frequency counter
    - For each literal:
        - Determine next block type for literals
        - Determine next tree index for literals
        - Add literal to the frequency counter
    - If the command has an explicit distance:
        - Determine next block type for distance codes
        - Determine next tree index for distance codes
        - Generate the distance code and add it to the frequency counter
5. Convert all symbol frequency counters into Huffman trees
6. Return the meta-block, and the final state object that can be fed into the next builder

Length & distance code generation can be optimized to various degrees. The most basic implementation could iterate all codes and check which values each can encode. The official implementation pushes for performance by combining patterns in the encodable values with arithmetic and bitwise operations. The custom implementation sits somewhere in the middle.

As with serialization, we can provide a parameter object — `BrotliCompressionParameters`. Here, it controls the generation of Huffman trees, and decides which distance code to pick if multiple valid options are available. By default, Huffman trees use the classic Huffman coding algorithm, and distance code picking will be explored in the next section.

Codes generated in the build process determine which codes become available in Huffman trees and how many bits they take. It is then up to serialization to look at each symbol, and pick the best code for it from the generated Huffman trees. This also applies to codes and values generated by block-switch builders.

4.5.2.1 Distance Code Picking Analysis

The distance code picker parameter is a function delegate called any time a distance can be written using two or more codes. It takes a list of candidate codes and frequencies of previously picked codes, and produces one code to add into the Huffman tree. Candidates are ordered by their position in the distance code symbol alphabet, which means that **Last** codes — those that refer to previous distances — always come first, and they are always followed by one either **Direct** or **Complex** code (both cannot refer to the same distance at once). We can try a few picking strategies:

- **First Option** is the default. It picks the first candidate, which is always a **Last** code.
- **Seen** picks the first candidate that had been picked before, or the first candidate if none have been picked before.
- **Frequent** picks the candidate that has been picked the most frequently so far, or the first candidate if none have been picked before.
- **Non-Last** picks the candidate which is not a **Last** code.

Figure 26 compares these strategies on the test corpus, using the **First** strategy as a baseline. Plot whiskers show the minimum and maximum.

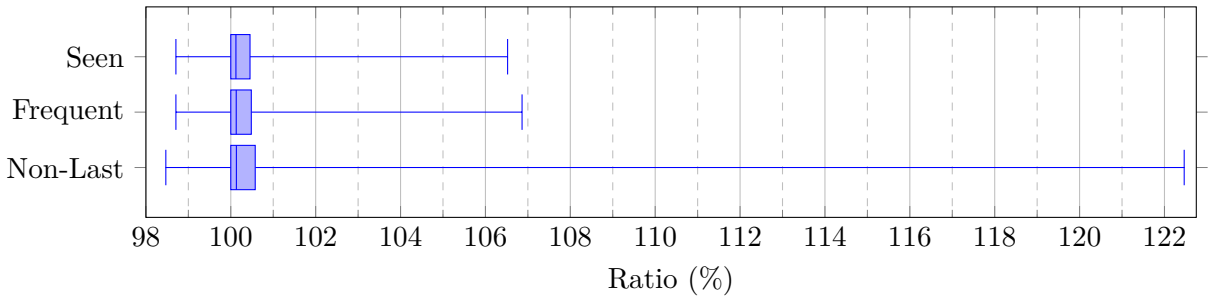


Figure 26: Comparison of distance code picking strategies on the test corpus.

4.5.3 Building Block-Switch Commands

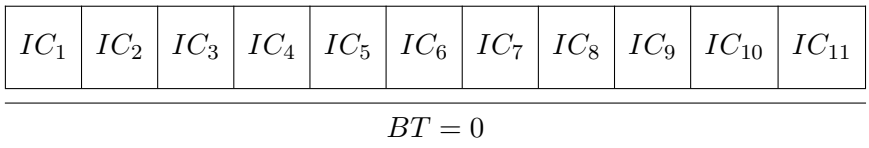
A `CompressedMetaBlockBuilder` allows configuring blocks and block-switch commands using a dedicated builder for each of the 3 categories of elements:

```
var literalBlocks      = builder.BlockTypes[Category.Literal];
var lengthCodeBlocks   = builder.BlockTypes[Category.InsertCopy];
var distanceCodeBlocks = builder.BlockTypes[Category.Distance];
```

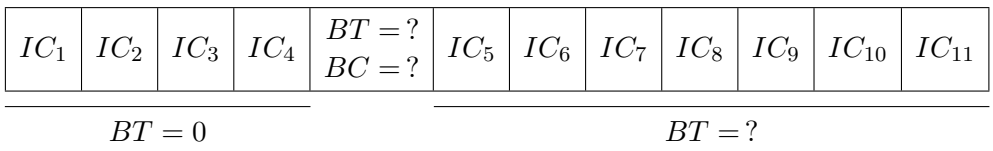
Listing 2: Obtaining block-switch builders from a `CompressedMetaBlockBuilder`.

The API is again best shown by example. The following listings and figures perform block splitting on a sequence of 11 insert&copy command length codes. The first block always has a block type of 0, and its length is set using either `SetInitialLength` or `AddBlock` with type 0. If `AddFinalBlock` is used, its length will cover all remaining symbols. Note that calling `Reset` is not strictly necessary, but it ensures we start from a clean slate when modifying a meta-block.

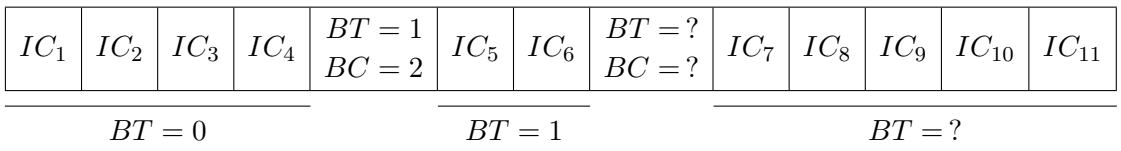
```
var blocks = builder.BlockTypes[Category.InsertCopy].Reset();
```




```
blocks.SetInitialLength(4);
```




```
blocks.AddBlock(type: 1, length: 2);
```




```
blocks.AddFinalBlock(type: 2);
```

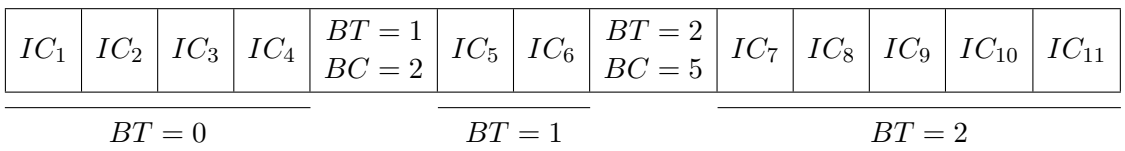


Figure 27: `BlockSwitchBuilder` API usage examples.

As mentioned in the previous section, block-switch commands are built when the meta-block is built. The build process for each block-switch builder works as follows:

1. Initialize two symbol frequency counters:
   - One counts block type codes
   - One counts block length codes, immediately counting a code that encodes the initial block length
2. Simulate all block-switch commands:
   - Generate the type code and add it to the frequency counter
   - If this is the last command and `AddFinalBlockSwitch` was used, calculate the final block length
   - Generate the length code and add it to the frequency counter

`BrotliCompressionParameters` again controls the generation of Huffman trees, and one parameter also picks the block type code if multiple can be used.

4.5.3.1 Block Type Code Picking Analysis

The block type code picker parameter works exactly like the distance code picker. Brotli defines two special codes — code 0 repeats the previous block type, code 1 increments the current block type and wraps around to $BT = 0$ if necessary.

The code picker is only asked if multiple codes can be used, so for a block type x the only possible candidate lists are $[0, x{+}2]$, $[1, x{+}2]$, or $[0, 1, x{+}2]$. The next test tries the following picking strategies:

- **Prefer Code 0** uses code 0 if available, otherwise uses the first candidate. This is the default strategy in the custom implementation.
- **Prefer Code 1** uses code 1 if available, otherwise uses the first candidate. This is how the official compressor behaves.
- **Non-Special** always picks the last candidate, avoiding both code 0 and 1.

Figure 28 compares these strategies on the test corpus, using **Prefer Code 0** as a baseline. Quality levels 0–3 do not perform block splitting and thus are excluded. Plot whiskers show the minimum and maximum.

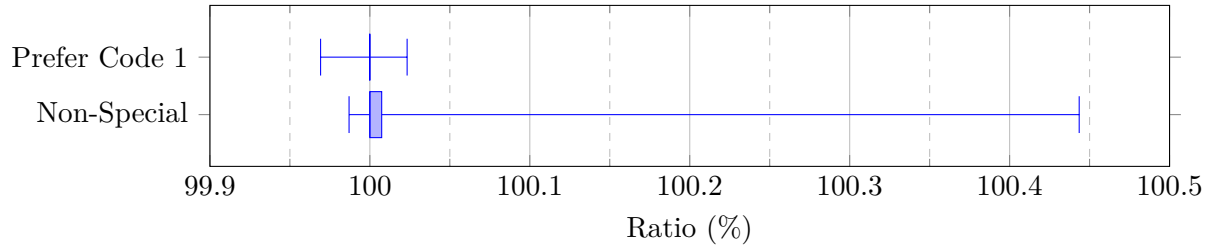


Figure 28: Comparison of block type code picking strategies on the test corpus.

4.5.4 Building Context Maps

This section introduces an API that simplifies context map creation. First, we must know the amount of block types, then construct either `ContextMapBuilder.Literals` or `.Distances`. Listing 3 showcases how the API can be used to set the mapping for entire block types, individual indices, or ranges of indices.[31] The context map array contents are shown below each API call, with asterisks indicating which positions were touched by that call.

```
// Assume the block types have already been setup.
Assert(builder.BlockTypes[Category.Distance].TypeCount == 3);

var contextMap = new ContextMapBuilder.Distances(blockTypeCount: 3);
// [ 0,0,0,0 | 0,0,0,0 | 0,0,0,0 ]

contextMap.Set(blockType: 0, values: new byte[]{ 0, 1, 2, 3 });
// [ 0,1,2,3 | 0,0,0,0 | 0,0,0,0 ]
//   * * * *

contextMap.Set(blockType: 0, index: 1, value: 0);
// [ 0,0,2,3 | 0,0,0,0 | 0,0,0,0 ]
//     *

contextMap.Set(blockType: 0, range: new IntRange(2, 3), value: 1);
// [ 0,0,1,1 | 0,0,0,0 | 0,0,0,0 ]
//       * *

contextMap.RepeatFirstBlockType(separateTreesPerBlockType: false);
// [ 0,0,1,1 | 0,0,1,1 | 0,0,1,1 ]
//             * * * *   * * * *

contextMap.RepeatFirstBlockType(separateTreesPerBlockType: true);
// [ 0,0,1,1 | 2,2,3,3 | 4,4,5,5 ]
//             * * * *   * * * *

// Build and assign.
builder.DistanceCtxMap = contextMap.Build();
```

Listing 3: `ContextMapBuilder` API usage examples.

[31] Note that calling `RepeatFirstBlockType` without separating the trees creates a useless context map where all block types have the exact same tree indices. It could however be used to propagate a pattern across all block types and then start changing individual indices.

The built `ContextMap` object stores the category (literals or distance codes), an immutable copy of the context map array, and the amount of Huffman trees calculated as $(1 + \textit{maximum value})$. For example, the context map from listing 3 would have 6 Huffman trees.

4.5.5 Final Comparison

This section applies the **rebuild** meta-block transformation to the test corpus. The transformation passes a meta-block to `CompressedMetaBlockBuilder` and simply rebuilds it, causing all Huffman trees and codes the be regenerated. Figure 29 shows size ratios between files originating from the official compressor and the rebuilt ones. Plot whiskers show the minimum and maximum. The second plot zooms in on quality levels 3–11.

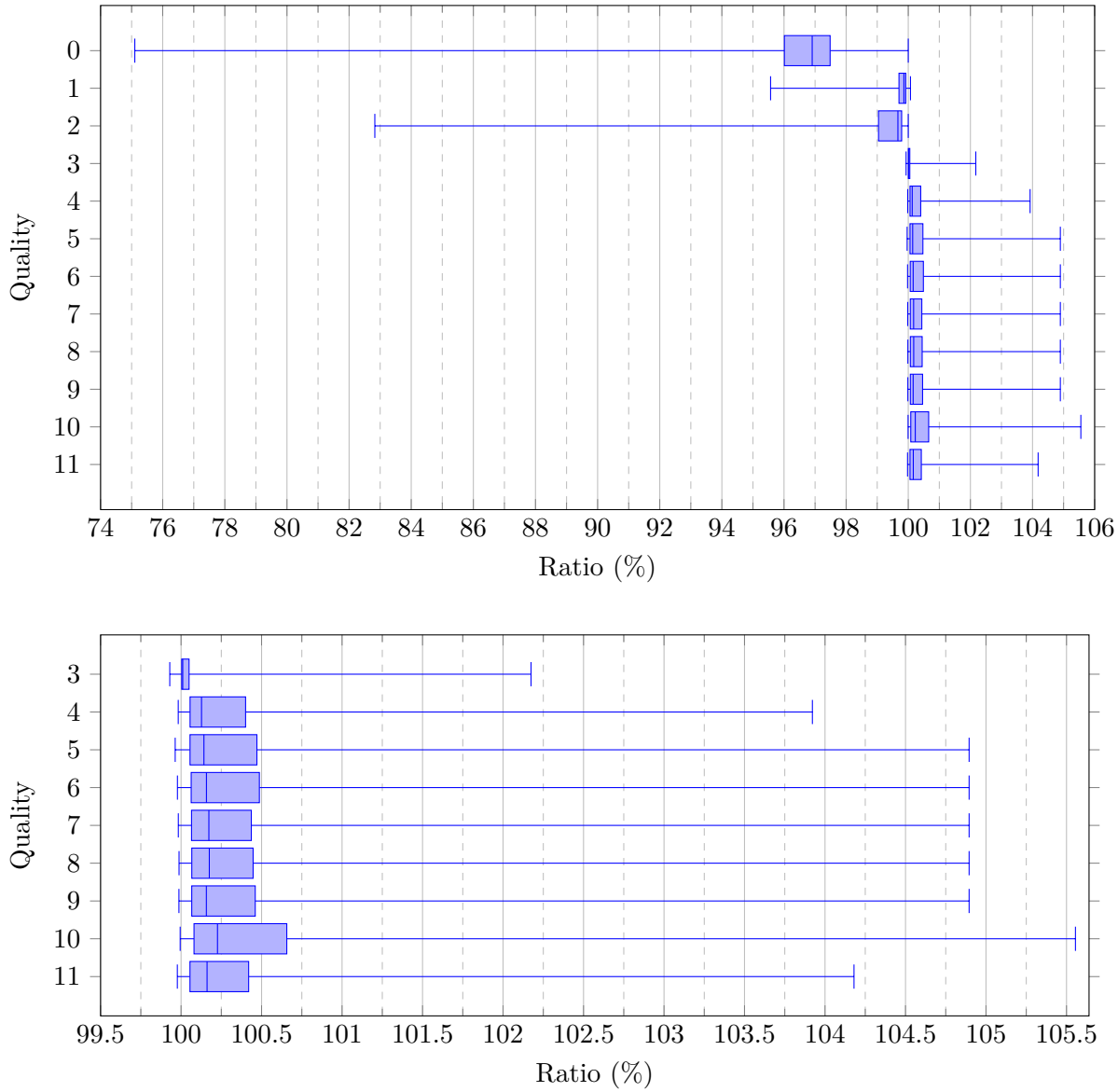


Figure 29: Compressed file size ratios (custom builder ÷ official implementation).

5 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. Finally, we will try experimenting with the source code in an attempt to find improvements.

5.1 Official Quality Levels

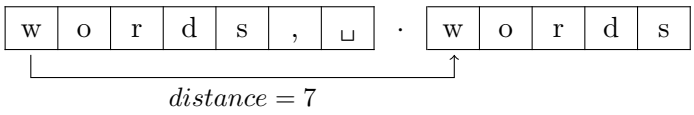
5.1.1 Quality Levels 0–1

The two lowest quality levels are reimplementations of Google's high speed Snappy[32] 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 30 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)

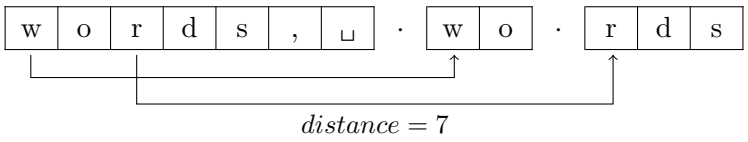


Figure 30: Insert&copy command generation pattern in official compressor's lowest quality levels.

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`)

[32] `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.

5.1.2 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.[33]
- **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[34] and level 6 in NGINX[35]), as they can compress dynamic web content reasonably fast while the highest quality levels may only be suitable for static content.

5.1.3 Quality Levels 10–11

The two highest quality levels are based on techniques used in Google's older Zopfli[36] compressor, combined with much more sophisticated use of the Brotli format features compared to previous quality levels.

[33] In quality level 4, the static dictionary is only enabled for files smaller than `1 MiB`.
[34] `https://httpd.apache.org/docs/2.4/mod/mod_brotli.html`
[35] `https://github.com/google/ngx_brotli`
[36] `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

5.2 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.

5.2.1 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 31 shows the difference in total sizes.

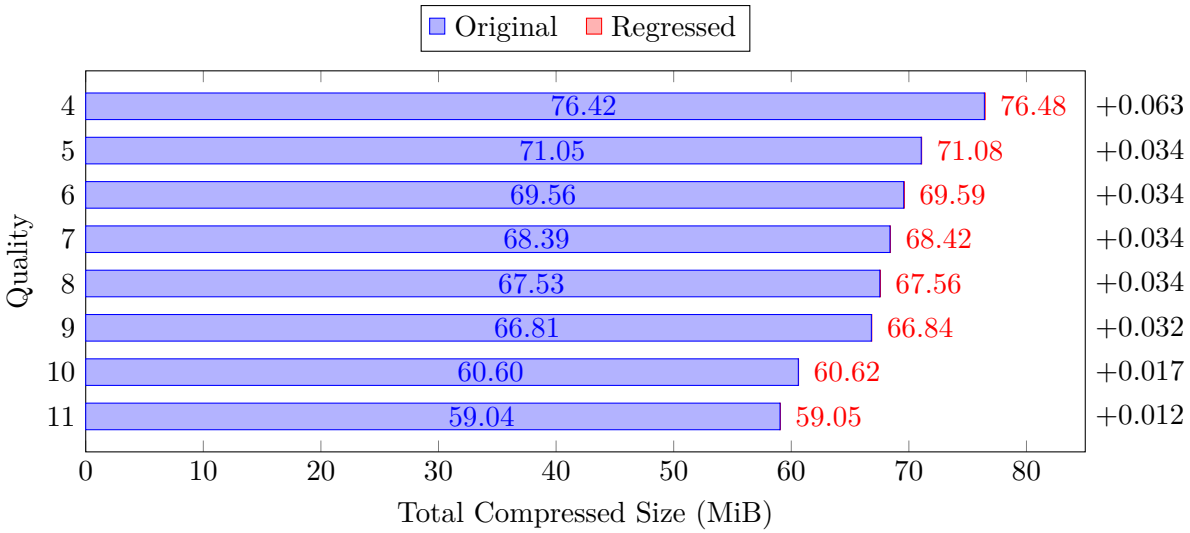


Figure 31: Test corpus size after disabling Huffman tree run optimization.

5.2.2 Evaluating Distance Parameters

The *postfix bits* parameter has a value of 0–3 and *direct code bits* has a value of 0–15,[37] 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.[38] 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 32 shows the total compressed sizes, and the change after the feature was turned off.

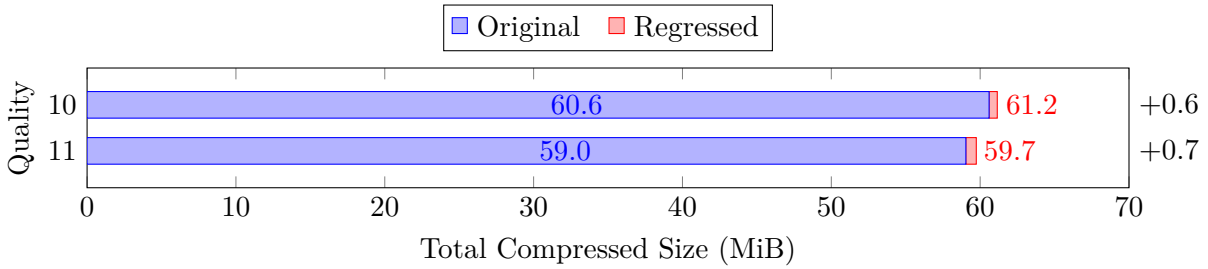


Figure 32: Test corpus size after disabling distance parameters.

5.2.3 Evaluating Static Dictionary

- **Medium quality levels (2–9)**[39]
  - 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.

[37] 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.

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

[39] 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)**[40]
  - 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.[41]

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 5, with sums on the bottom showing us how many dictionary references each quality level has produced in total.

Table 5: 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

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

[41] 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 6 shows the % of bytes generated by dictionary references in English plain text files.

Table 6: 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 33 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.

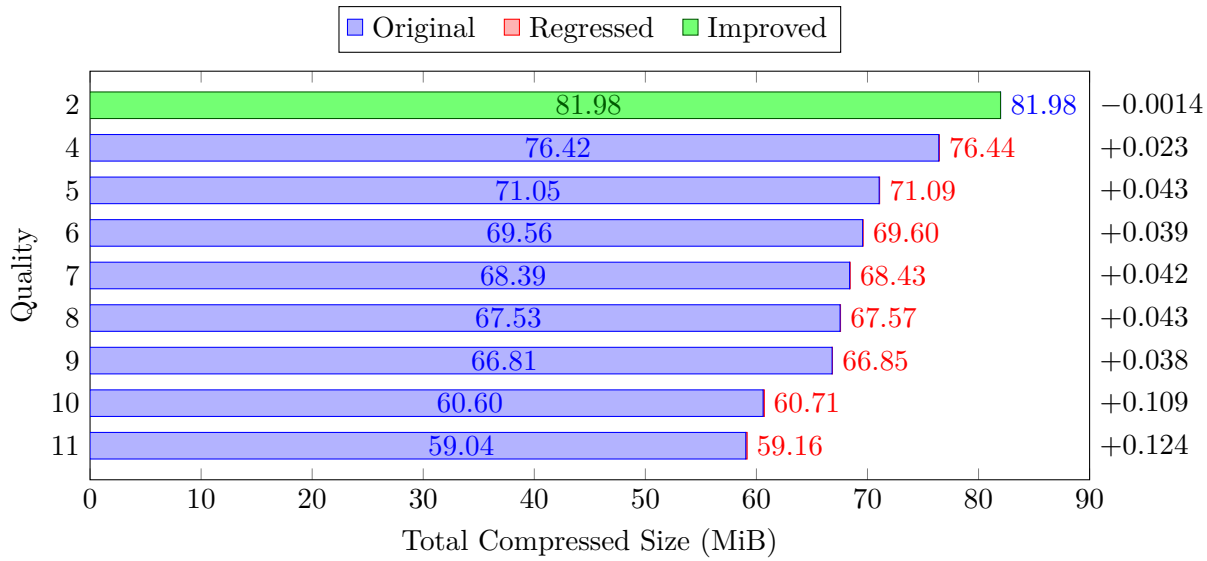


Figure 33: Test corpus size after disabling the static dictionary.

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.

5.2.4 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.

5.2.4.1 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.[42]

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.[43]

If the compressor decides to use context modeling, it heuristically picks one of 3 preset context map patterns.[44] Table 7 shows how many times each pattern emerged in meta-blocks in the test corpus.

Table 7: 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.[45] 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.

[42] Source file `metablock.c` (`BrotliBuildMetaBlockGreedyInternal`).
[43] Source file `encode.c` (`ChooseContextMode`).
[44] Source file `encode.c` (`DecideOverLiteralContextModeling`).
[45] Same mechanism as `RepeatFirstBlockType(true)` in the context map builder API.

5.2.4.2 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.[46] 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.[47]

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:[48]

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 34 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.

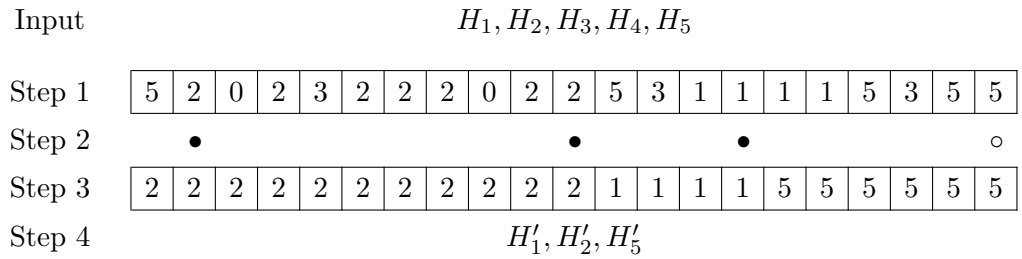


Figure 34: Example of one iteration of official compressor's advanced block splitting algorithm.

The resulting histograms go through a merging process, which reduces their amount by repeatedly finding 2 most similar histograms, and merging them into one.[49] 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.

[46] Source file `encode.c` (`ChooseContextMode`).
[47] Source file `block_splitter.c`.
[48] Source file `block_splitter_inc.h` (`FindBlocks`).
[49] Source file `block_splitter_inc.h` (`ClusterBlocks`).

Figure 35 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.

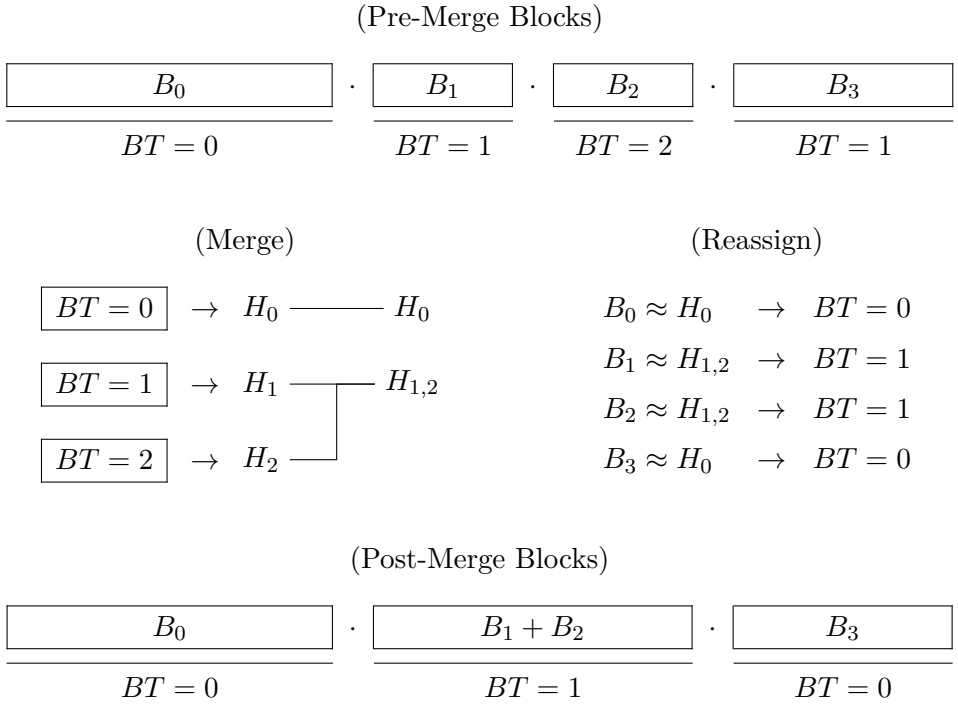


Figure 35: Example of how the block splitter could merge and reassign blocks generated by the previous step.

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.[50]

Figure 36 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.

[50]Source file `cluster_inc.h`.

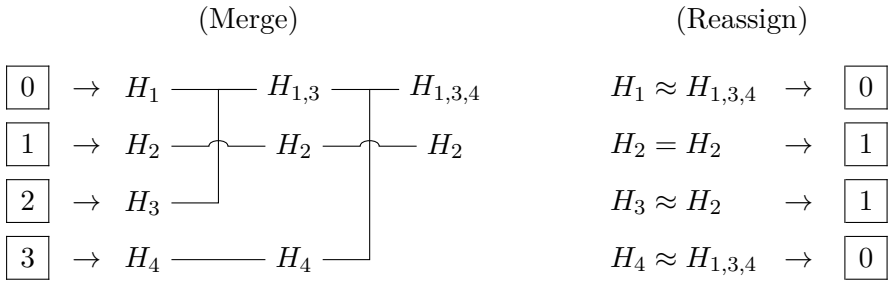


Figure 36: Example of how a distance context map could be created by the official compressor.

5.2.4.3 Evaluation

Figure 37 shows the total size difference after disabling context modeling, and instead assigning each block type a unique Huffman tree. Figure 38 turns off both features at once.

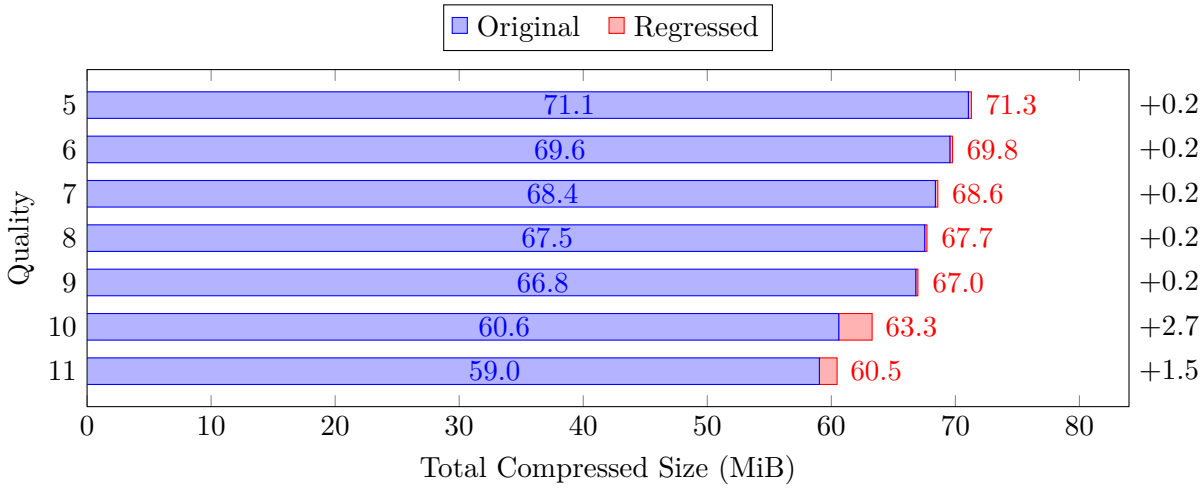


Figure 37: Test corpus size after disabling context modeling for both literals & distance codes.

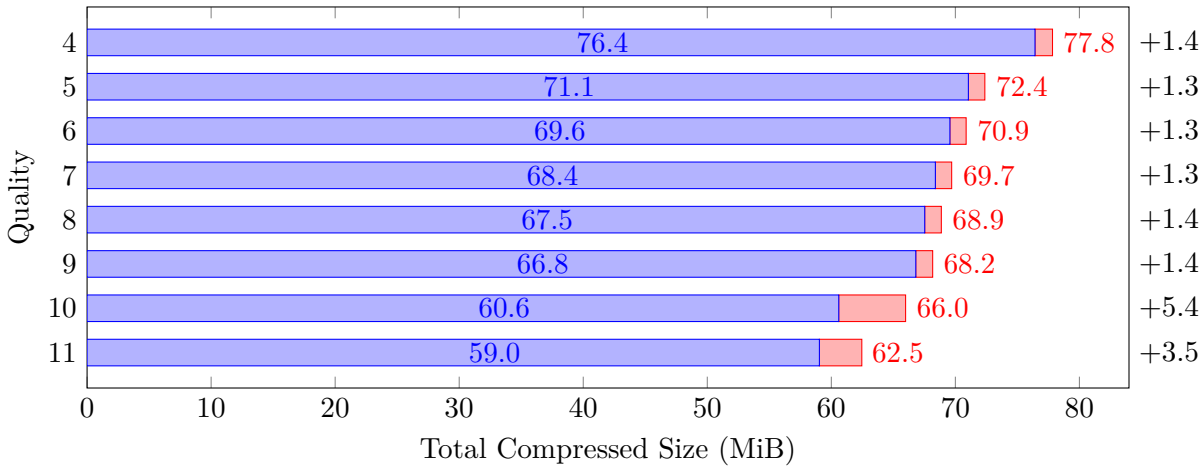


Figure 38: Test corpus size after disabling both block splitting and context modeling.

5.3 Modifications to the Official Compressor

The final section attempts to find possible improvements in the official compressor by modifying its source code.

All modified versions of the official compressor are compiled using the configuration described in section 2.2, and compared to baseline results from an unmodified executable compiled in the same way.

Besides comparing the compression ratio, these experiments will also test compression speed to get an idea about the cost of each modification versus its size savings. The benchmark process involves compressing the entire corpus several times, and taking an average of the total compression times. Figure 39 shows the baseline for compression times.

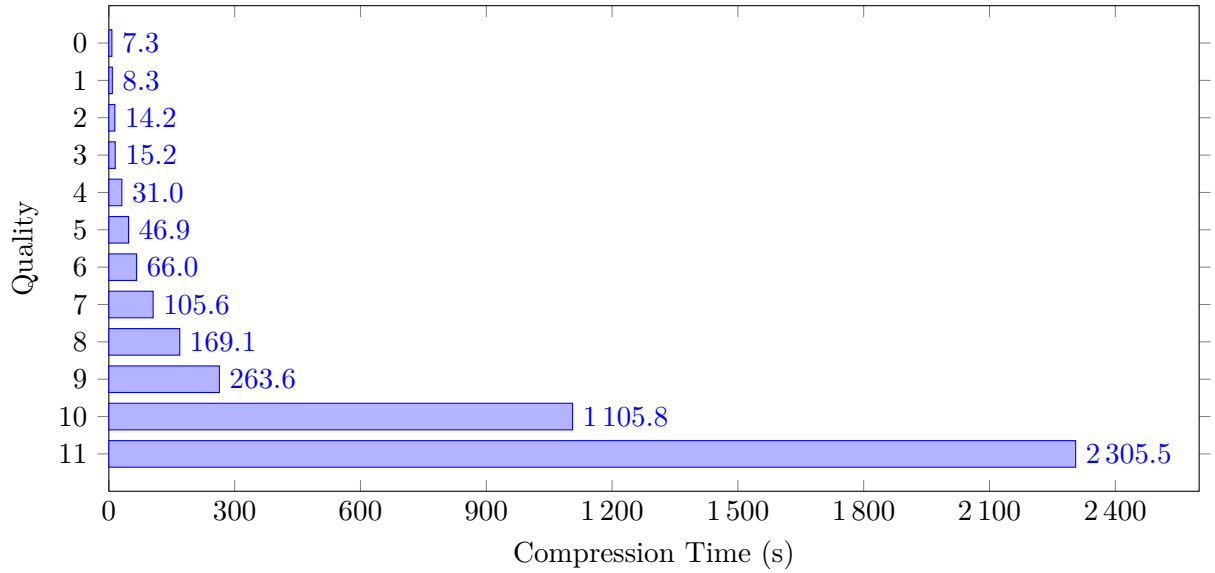


Figure 39: Total corpus compression time before applying any modifications from this section.

5.3.1 Modification #1: Dictionary Lookup in Medium Quality Levels

The first experiment concerns the limited dictionary lookup used by quality levels 2 and 4–9. Its intention is to increase the amount of words usable by the lookup procedure. We can think of several possible approaches:

- Incorporate more word transforms, inflicting a performance penalty on each lookup
- Increase hash range from 14 bits to 15 bits, inflating the executable by `96 KiB` ($\approx 12\%$)
- Increase bucket size from 2 to 3, inflating the executable by `48 KiB` ($\approx 6\%$) with a performance penalty on each lookup
- Use a completely different hash table implementation or hashing function

The first option was chosen for this experiment. Out of the 121 transforms, 43 can be categorized as **identity + suffix** transforms. These types of transforms have several advantages — they

only include a 1–7 byte suffix without modifying the original word, they do not require any additional lookups, and they incur a performance penalty on lookup only when that lookup is successful — making them a good candidate for this experiment.

The dictionary lookup and command processing systems in medium quality levels were not designed to process transforms which increase the word length. Simply adding support for this case increased compression time by ≈ 8% over the baseline (averaged per quality level).

Several tests were conducted to find the best results, balancing savings from fewer literals and losses from having to encode large distances. After each test, the entire corpus was compressed, and for each quality level the total size after the modification was compared to the total size before the modification.

1. Checking all 43 suffixes reduced total compressed size by an average of ≈ 0.0018% per quality level.
2. Checking the 34 suffixes of length 2 and higher reduced total size by ≈ 0.0016%.
3. Checking the 29 suffixes with transform IDs 1–60 reduced total size ≈ 0.0018%.
4. Checking the 29 suffixes for which $(ID \div length \leq 20)$ reduced total size by ≈ 0.0025%.
5. Checking the 14 suffixes for which $(ID \div length \leq 10)$ reduced total size by ≈ 0.0026%.

The best result also increased compression time by ≈ 5% over the baseline, which is in fact less than the 8% found after merely implementing support for suffix processing in the medium quality levels.

Table 8 lists the increases in compression time and reductions in total compressed size. The table includes compression time before and after adding the suffix checks — note again that some quality levels show worse results while doing less work, possibly because of a strange interaction with compiler optimizations.

Table 8: Results of implementing dictionary suffix lookup in quality levels 2 and 4–9.

Quality	Time (no checks)	Time (with checks)	Size
2	+0.6%	+0.9%	−0.00006%
4	+3.1%	+7.3%	−0.0005%
5	+6.8%	+4.1%	−0.0037%
6	+8.0%	+5.0%	−0.0065%
7	+10.1%	+5.8%	−0.0006%
8	+13.1%	+6.2%	−0.0026%
9	+14.8%	+5.9%	−0.0038%

Overall, the performance penalty is not worth the tiny improvement in compression size. While it might be possible to improve medium quality levels in other ways, the changes needed to make this particular experiment work proved to have too many disadvantages.

5.3.2 Modification #2: Advanced Block Splitter Seeding Strategy

The block splitting algorithm for high quality levels uses an iterative approach, whose initial state — seed — is generated by a pseudorandom sample of the symbols. This experiment tries an alternative seeding strategy.

The idea is to take the greedy block splitting algorithm, which is used by medium quality levels, apply it to the symbol sequence, and use its block types as histograms for the seed. The greedy block splitter has several parameters which can be tweaked for each of the 3 categories.

We will try activating the new strategy separately for each category, and without changing any of the parameters. With the established baseline, we can then focus on improving each category individually.

The first attempt resulted in a nearly universal reduction of block types and Huffman trees across meta-blocks in the test corpus, but not necessarily in the compressed size. Table 9 shows how much the total compressed size and amount of block types per meta-block has increased or decreased after the new strategy was applied.

Table 9: Results of first attempt at using medium quality block splitter to seed the high quality block splitter.

	Total Compressed Size			Block Type Count		
Quality	[L]	[I]	[D]	[L]	[I]	[D]
10	+0.033%	+0.068%	+0.041%	−4.8%	−9.7%	−8.8%
11	−0.002%	−0.006%	+0.069%	−5.4%	−3.3%	+0.8%

Because each step of the iterative algorithm can only reduce or keep the same amount of block types it is given at the start, the ratios suggest that the greedy algorithm is not generating enough of them. The greedy algorithm parameters control how many symbols are consumed before a decision whether to split is made, and set a threshold that needs to be crossed in order to start a completely new block type. Table 10 shows the best results after tweaking each parameter.

Table 10: Results of second attempt at using medium quality block splitter to seed the high quality block splitter.

	Total Compressed Size			Block Type Count		
Quality	[L]	[I]	[D]	[L]	[I]	[D]
10	+0.029%	+0.016%	−0.006%	−7.9%	+0.4%	+1.7%
11	−0.002%	+0.012%	−0.004%	−10.0%	+3.7%	+1.9%

The only overall improvement was in the distance code category, but even within that category there is a considerable amount of files which became larger. Considering both the mixed results and a $\approx 25\%$ increase in compression time, this experiment can be considered unsuccessful.

5.3.3 Modification #3: Forcing Literal Context Modes

Quality levels 10 and 11 are the only ones which generate context maps based on the input data, rather than pick a statically defined one, however they are still limited to two literal context modes — `UTF8` and `Signed`. To test whether another mode could be more efficient, and how well the current heuristic that chooses the mode works, we will compress the entire test corpus once for each literal context mode, forcing the same mode to be applied to all files.

Although a large part of the test corpus focuses on website resource files, which usually use the UTF-8 encoding and thus should work best with the `UTF8` mode, we may find discrepancies in the Canterbury and Silesia corpora that include a variety of file formats. Compressing the entire corpus with each of the literal context modes reveals a few general insights:

- Out of all 169 files:
  - 158 files used `UTF8`
  - 10 files used `Signed`
  - 1 file had no compressed meta-blocks
- If we only considered these 2 modes in quality level 10:
  - 17 files would in total save `15 089 B` if they chose `Signed` instead of `UTF8`
  - 1 file would save `74 B` if it chose `UTF8` instead of `Signed`
- If we only considered these 2 modes in quality level 11:
  - 20 files would in total save `10 341 B` if they chose `Signed` instead of `UTF8`
  - 2 files would in total save `3 006 B` if it chose `UTF8` instead of `Signed`

This suggests the heuristic for picking between `UTF8` and `Signed` modes generally works well. Most files that chose *wrong* are small, and so are the differences in size after compressing them with the *correct* mode. If we dig deeper, we can find more interesting takeaways regarding specific files:

- Both Chinese Bible translations benefited from `LSB6` mode, but only in quality level 10. The savings were $\approx 1.5\%$ for Simplified Chinese and $\approx 1.3\%$ for Traditional Chinese.
- Some of the binary files (`Canterbury/ptt5`, `Silesia/sao.bin`, `Silesia/ooffice.dll`, `Snappy/kppkn.gtb`) compressed better with `LSB6`/`MSB6`/both than with the `Signed` mode.
- From the website sub-corpus, two files with the most significant gains from using `LSB6` or `MSB6` contained a considerable amount of Base64 encoded data. This prompted further investigation into Base64 in section 5.3.3.1.

Table 11 assesses for how many files from each part of the test corpus a particular literal context mode yielded best (or tied) results, and if every file chose its best mode, how many bytes would be saved (and by how many % it would reduce total compressed size). Files with no literal context maps were omitted.

Table 11: Results of compressing the entire test corpus once for each literal context mode.

		Best/Tied Literal Context Mode				
	Quality	`LSB6`	`MSB6`	`UTF8`	`Signed`	Potential Savings
Canterbury Corpus	10	3 / 0	1 / 0	6 / 0	1 / 0	`1 831 B` (≈ 0.36%)
	11	1 / 1	0 / 1	6 / 0	3 / 1	`1 200 B` (≈ 0.25%)
Silesia Corpus	10	0 / 0	2 / 0	6 / 0	4 / 0	`35 907 B` (≈ 0.07%)
	11	1 / 0	1 / 0	6 / 0	4 / 0	`15 976 B` (≈ 0.03%)
Snappy Test Data	10	0 / 0	2 / 0	4 / 0	0 / 0	`277 B` (≈ 0.07%)
	11	0 / 0	2 / 0	4 / 0	0 / 0	`349 B` (≈ 0.09%)
Bible Languages	10	6 / 0	0 / 0	1 / 0	0 / 0	`31 306 B` (≈ 0.44%)
	11	3 / 0	0 / 0	4 / 0	0 / 0	`494 B` (≈ 0.01%)
Website Resources	10	2 / 1	8 / 3	105 / 1	7 / 4	`779 B` (≈ 0.02%)
	11	2 / 3	6 / 7	97 / 10	12 / 10	`993 B` (≈ 0.02%)

Although the savings may not appear high, if we only consider files which do benefit from different modes, their individual savings range from ≈ 0.003% to ≈ 11.9%, and average ≈ 1.3%.

5.3.3.1 Investigating Base64

Base64 is a form of encoding that represents binary data using an alphabet of 64 symbols — 52 lower case and upper case letters, 10 digits, and 2 additional symbols that vary with the use case. The end of a Base64 string may include padding in the form of '=' characters. In websites, Base64 may be used to embed binary files, such as images or fonts, into CSS or JS resources.[9]

The previous experiment suggests that `LSB6` could work well for Base64 encoded data. If we confirm that hypothesis and are able to efficiently identify files containing such data, it could result in occasional but useful gains in HTTP compression.

For the first point, the entire test corpus was converted into Base64 and compressed using each of the 4 literal context modes. Nearly every single file compressed best with the `LSB6` mode, the only meaningful exception being `Silesia/x-ray.dicom` which in quality level 11 compressed worse by ≈ 0.07% than `UTF8`. Figures 40 and 41 show total sizes for both quality levels.

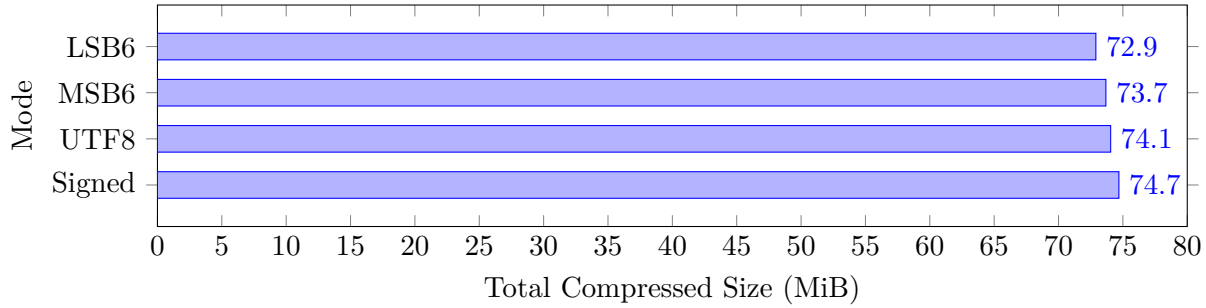


Figure 40: Test corpus size after converting all files to Base64, and compressing them with each literal context mode using quality level 10.

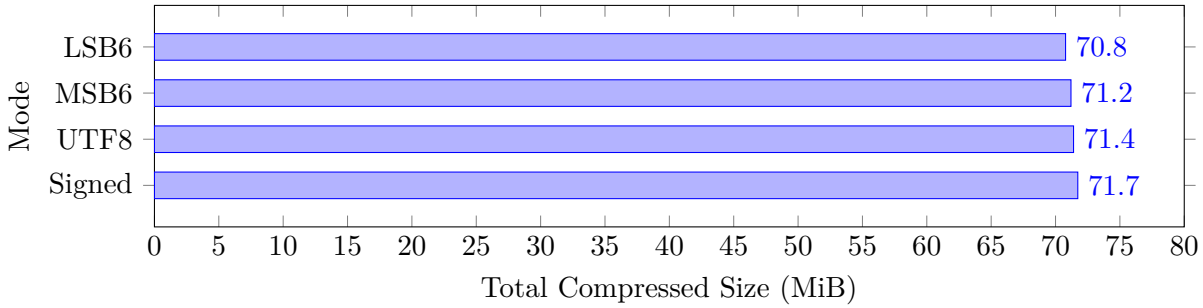


Figure 41: Test corpus size after converting all files to Base64, and compressing them with each literal context mode using quality level 11.

Websites use a Base64 variant with '+' and '/' as the two special symbols. In a UTF-8 encoded file, the full symbol set is represented by the following 1-byte code point ranges: 43 (plus), 47 (slash), 48–57 (digits), 61 (equals), 65–90 (upper case letters), and 97–122 (lower case letters).

Although 22 files from the website sub-corpus contain one or more Base64 encoded sections, at this point we can only set literal context modes for entire meta-blocks. Modifying the compressor to use `LSB6` when the UTF-8 heuristic succeeds and at least 95% of the data matches the listed 1-byte code points resulted in changes to two files:

- A `39.7 KiB` file compressed better, by ≈ 1.1% and ≈ 0.7% with quality levels 10 and 11 respectively.
- A `643 KiB` file compressed worse, by ≈ 0.004% and ≈ 0.01% with quality levels 10 and 11 respectively.

Adding a thorough Base64 detection test, based on the existing UTF-8 detection test in the source code, increased compression time by ≈ 7.6% and ≈ 3.3% for quality levels 10 and 11 respectively. We could simplify the test in several ways, such as:

1. Instead of testing all code point ranges separately, we could accept all 1-byte code points between 43–122 as Base64. To counteract false positives, the minimum % had to be increased from 95% to 98.5%. This reduced the increases in compression time from ≈ 7.6% to ≈ 4.6%, and from ≈ 3.3% to ≈ 2.1%.
2. Eschew proper UTF-8 code point processing and treat every byte as a 1-byte code point. This needed no additional minimum % adjustments, and reduced the increases in compression time from ≈ 7.6% to ≈ 3.2%, and from ≈ 3.3% to ≈ 1.7%.

Section 5.3.4 looks at the possibility of using different literal context modes for different block types. We will revisit Base64 under the assumption that the block splitter can separate Base64 data into its own block types.

5.3.4 Modification #4: Per-Block-Type Literal Context Modes

As mentioned before, the format allows setting a different literal context mode for each block type, however the official compressor does not take advantage of that feature. This experiment proposes a way to use the existing heuristic that chooses between `UTF8` and `Signed` mode on a per block type basis.

We want to assign literal context modes after block splitting, but before context modeling. The following assignment strategy was devised:

1. For every input byte, determine if it is part of a UTF-8 code point.[51]
2. Iterate all insert&copy command literals, keeping track of the current block type.
3. Count how many literals belong to each block type, and how many of those are also part of a UTF-8 code point.
4. Calculate the % of UTF-8 literals in each block type.
5. If a block type's % is greater than a set minimum, assign it the `UTF8` mode, otherwise assign it the `Signed` mode.

The original heuristic analyzes the input data before it becomes a sequence of insert&copy commands, and chooses `UTF8` if it deems at least 75% of it to be UTF-8 encoded.

Reusing 75% as the minimum in this new algorithm turns out to have largely negative consequences — many block types get mislabeled as `UTF8` despite `Signed` compressing them better. It took increasing the minimum to $\approx 99\%$ to stop the mislabeling — in the test corpus, 99.1% led to satisfactory results that are shown in table 12. Both quality levels saw size improvements in 5 files and regressions also in 5 files, however the regressions averaged less than 0.0005% per file making them negligible.

Table 12: Changes in total compressed corpus size for various constants for choosing per block type literal context modes.

Quality	$> 75\%$	$> 99.1\%$
10	+0.162%	−0.026%
11	+0.024%	−0.021%

Unlike previous modifications, this one actually sped up compression by $\approx 3.3\%$ in quality level 10, and by $\approx 0.98\%$ in quality level 11.

[51] Implementation of this experiment used a *bit array* to assign a boolean value to each byte of the input, increasing memory use by one eighth of the input size.

5.3.4.1 Revisiting Base64

The final experiment includes a version of the Base64 test from section 5.3.3.1 in the new literal context mode picking algorithm. The following changes were made:

- When counting literals:
  - If a block type counts a UTF-8 literal, and that literal matches one of the Base64 code points, increment a separate Base64 counter for that block type.[52]
- When assigning a literal context mode to a block type:
  - If the ratio of UTF-8 literals exceeds the previously established minimum of 99.1%, and at the same time the ratio of Base64 literals to all counted literals exceeds 99.5%, use `LSB6` mode instead of `UTF8` mode.

The results were overall slightly improved, with table 13 listing the changes in total size.

Table 13: Changes in total compressed corpus size with Base64 test per block type.

Quality	Before	After
10	−0.0255%	−0.0261%
11	−0.0209%	−0.0213%

If we only look at the website sub-corpus, which was the point of interest for this experiment, we find overall improvements by ≈ 0.0013% in quality level 10, and by ≈ 0.0008% in quality level 11. Out of 132 files, of which 22 files contained one or more Base64 encoded sections:

- 13 files contained Base64 and also used `LSB6` in one or more block types
- 9 files contained Base64 but did not use `LSB6`
- 4 files did not contain Base64 and used `LSB6` in one or more block types

Table 14: Statistics of website sub-corpus file improvements/regressions when using `LSB6` mode.

(Quality 10)

	Improved	Regressed
With Base64	5	5
No Base64	0	1

(Quality 11)

	Improved	Regressed
With Base64	5	5
No Base64	1	3

With the updated algorithm, compression was sped up by ≈ 3.4% in quality level 10, and by ≈ 1.9% in quality level 11 when compared against the baseline. The experiment shows potential that could be developed further, but it might call for a new block splitting algorithm designed with literal context modes in mind.

[52] This strategy is similar to the optimization that treated all bytes as 1-byte code points, however in this case we know which bytes are not UTF-8 and are able to skip them.

6 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[2] 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.

Utility applications based on the custom implementation proved to be very helpful when studying features of the Brotli format — how they are used by different quality levels, and how they were affected by the various experiments with both the format and the official source code. The object representation made it easy to collect statistics about each element of the format, which were used to create many of the figures and tables included in the thesis.

The process of developing the custom implementation prompted questions regarding possible serialization and code picking strategies in various parts of the format. Many different strategies and heuristics were tried on the test corpus, and compared against those used by the official compressor implementation. Sometimes it would reveal a potential for small improvements, other times it would show highly varying results that reaffirm the fact a single strategy almost never works equally well on all possible inputs.

The final goal of this thesis was to design and implement modifications compatible with the Brotli format, and compare their compression size and speed to the official implementation. In order to find where the official implementation could be improved and how it balanced the two compression performance metrics, it was important to (1) look at the differences between quality levels and their real use cases, and (2) understand how exactly were key parts of the format implemented. The thesis explored these parts of the official implementation in vast detail, identifying a few areas where the format was not used to its full potential. The modifications themselves had mixed results; 2 out of 4 modifications — those targeting context modeling for literals — demonstrated ideas that could be developed further, but even in their current form led to reasonable size savings and in one case a reduction in compression time.

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] 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.

[3] 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.

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

[5] 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.

[6] 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.

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

[8] Charles Bloom. *Length-Limitted Huffman Codes Heuristic*. 2010-07-03. URL: https://cbloomrants.blogspot.com/2010/07/07-03-10-length-limitted-huffman-codes.html.

[9] MDN Contributors. *Data URLs*. 2020-03-03. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Data_URIs.