Brotli: A general-purpose data compressor

JYRKI ALAKUIJALA, Google Research
ANDREA FARRUGGIA, Università di Pisa
PAOLO FERRAGINA, Università di Pisa
EUGENE KLIUCHNIKOV, Google Research
ROBERT OBRYK, Google Research
ZOLTAN SZABADKA, Google Research
LODE VANDEVENNE, Google Research

`Brotli` is an open-source general-purpose data compressor introduced by Google in late 2013, and now adopted in most known browsers and Web servers. It is publicly available on GitHub and its data format was submitted as RFC 7932 in July 2016 . `Brotli` is based on the Lempel-Ziv compression scheme and planned as a generic replacement of Gzip and ZLib. The main goal in its design was to compress data on the Internet, which meant optimizing the resources used at decoding time, while achieving maximal compression density.

This paper is intended to provide the first thorough, systematic description of the Brotli format as well as a detailed computational and experimental analysis of the main algorithmic blocks underlying the current encoder implementation, together with a comparison against compressors of different families constituting the state-of-the-art either in practice or in theory. This treatment will allow us to raise a set of new algorithmic and software engineering problems that deserve further attention from the scientific community.

CCS Concepts: • **Theory of computation** → *Data compression*; *Pattern matching*; *Problems, reductions and completeness*; *Shortest paths*; • **Mathematics of computing** → *Coding theory*; • **Information systems** → *Information storage systems*; *Storage management*;

Additional Key Words and Phrases: Data Compression, Lempel-Ziv parsing, Treaps, NP-completeness, Shortest Paths, Experiments

ACM Reference format:
Jyrki Alakuijala, Andrea Farruggia, Paolo Ferragina, Eugene Kliuchnikov, Robert Obryk, Zoltan Szabadka, and Lode Vandevenne. 2017. Brotli: A general-purpose data compressor. *ACM Trans. Comput. Syst.* 0, 0, Article 0 (2017), 32 pages.
https://doi.org/0000001.0000001_2

This work has been partially supported by a Google Faculty Research Award, winter 2016. Andrea Farruggia is currently at Google, Zurich (Switzerland).
Author's addresses: P. Ferragina, Dipartimento di Informatica, Largo B. Pontecorvo 3, 56127 Pisa, Italy; all other authors are with Google, Brandschenkestrasse 110, 8002 Zürich, Switzerland.

0734-2071/2017/0-ART0 $15.00
https://doi.org/0000001.0000001_2

1 INTRODUCTION

In the quest for the ultimate data compressor, algorithmic theory and engineering go hand in hand. This point is well illustrated by the number of results and implementations originated by the fundamental achievements due to Lempel and Ziv (at the end of the 1970s) and to Burrows and Wheeler (at the end of the 1980s), who introduced two main paradigms for efficaciously compressing arbitrary data [21, 27]: namely, dictionary-based compression [28, 29] and BWT-based compression [4, 11]. Algorithms are known in both families that require theoretically linear time in the input size, both for compressing and decompressing data, and achieve compressed space that can be bound in terms of the k-th order empirical entropy of the input.

However, theoretically efficient compressors are either optimal in obsolete models of computations (like the RAM model) and thus turn out to be slow in practice, or they solve problems that are somehow far from the ones arising in real applications. The latter, increasingly, are adding new algorithmic challenges arising from new networking and hierarchical-memory features, as well as the very diverse entropy sources involved in the compression process, such as text documents, source code, structured data, images, and a variety of other data types. As a result, the compressors running behind real-world large-scale storage systems are very sophisticated in that they combine some of these theoretical achievements with several heuristics and technicalities that are often hidden in the software and whose algorithmic motivations and performance are difficult to discover or prove.

At a W3C meeting in late 2013, Google introduced `Brotli`,[1] which was initially designed as a compressor for Web fonts[2], then extended to be an HTTP compressor (September 2015). Its data format was submitted as RFC 7932 in July 2016 [1]. `Brotli` is an open-source data compression library[3] developed by Jyrki Alakuijala and Zoltán Szabadka (two of the authors of this paper) based on the Lempel-Ziv compression scheme and intended as a generic replacement of Gzip and ZLib. The main goal in Brotli's design was to achieve maximal compression density, even on very short files (around 55kB), based on a *pseudo*-optimal entropy encoding of LZ77-phrases that does not slow down the decoding speed from that of Gzip and `deflate`. Specific attention was devoted to compressing data on the Internet, which meant optimizing the resources used at decoding time, be they the memory necessary for the backward reference window or the memory consumption due to the several compressed transfers open at the same time by a browser. These requirements forced us to inject into Brotli many novel algorithmic solutions that make it very different from all the other compressors in the LZ-family, such as: (i) hybrid parsing of the input text in LZ-phrases and long runs of literals driven by a pseudo-optimal Shortest Path scheme derived from `Zopfli` (a compressor previously proposed by Google [26][4]); (ii) Huffman encoding of LZ-phrases based on second-order models, re-use of entropy codes, and relative encoding of distances to take advantage of locality of references; (iii) optimization of the number of Huffman models available and the proper block splitting, based on clustering symbol distributions, in order to further reduce the compressed space and the cache-misses at decompression time; (iv) a static dictionary (composed of strings of variable length, along with a set of transformations that can be applied to such strings) constructed to improve the compression of small files.

[1] Brotli is a Swiss German word for a bread roll and literally means "small bread".
[2] See WOFF 2.0 at `https://www.w3.org/TR/2016/CR-WOFF2-20160315/`.
[3] See `https://github.com/google/brotli`.
[4] Zopfli is a Swiss German word for a braided sweet bread and literally means "little plait".

Because of its performance Brotli is currently adopted in most known browsers – Google Chrome (since version 50), Microsoft Edge (since version 15), Mozilla Firefox (since version 44), Opera (since version 38) – and in Web servers, such as Apache HTTP Server and Microsoft IIS. For an updated picture of this usage we refer the reader to the CanIUse site[5].

However, apart from the public availability of Brotli's code on GitHub and a few technical posts that have appeared on the Web, which sketch the main features of the compressor and provide some figures on its overall performance, no detailed description of Brotli's inner algorithmic choices and workings have been "disclosed" to the public, probably because of its *sophisticated* design choices, which are difficult to derive directly from the code. This paper is intended to fill this scientific gap by offering the first thorough, systematic description of the Brotli format, as defined in RFC7932 [1], as well as an analysis of the main algorithmic blocks underlying the current encoder implementation. On the algorithmic side, particular attention will be devoted to study the highest quality compression (range 10–11), which is best suited for *compress-once-decompress-many* needs. In this respect we will analyze the complexity of the LZ77-parsing problem dealt with in Brotli and show that one of its variants is $\mathcal{NP}$-hard. We will also discuss analytically and experimentally the efficacy of the heuristic proposed to solve it by comparing its result against the one computed by another recent compression scheme, named Bc-Zip [9, 10], which is based on an optimization approach that achieves performance guarantees in trading decompression speed for compressed space occupancy. On the experimental side, we will supplement the algorithmic analysis of Brotli's components with an evaluation of their individual impact on its final performance, together with a comparison against compressors of different families constituting the state-of-the-art either in practice or in theory. The ultimate take-away message of this paper will be a set of algorithmic problems that deserve further attention from the scientific community because their efficient solution could improve Brotli, shed more light on its efficacious design choices, and eventually lead to new compression tools.

This paper is organized as follows. Section 2 offers a high-level description of the algorithmic structure of Brotli by identifying two main phases that are detailed in Section 3 and Section 4. In the former section we state the problem of computing a good entropic LZ77-parsing of the input data; show in Appendix A that a significant variant is $\mathcal{NP}$-hard; then detail the algorithm and its corresponding data structures devised in Brotli to provide an approximate solution in efficient time (Subsection 3.1). We will devote special attention to describing the use that Brotli makes of a *dynamic Treap* in order to find the *"best"* LZ77-phrases that are then used to populate a *weighted DAG* over which a "special" *shortest path computation* will be performed. That shortest path will provide an LZ77-parsing whose compressed-space performance will be evaluated in Subsection 3.2 by experimentally comparing it against the *principled* LZ77-parsing produced by Bc-Zip. This comparison will show the pros and cons of each one of those strategies and indicate, on the one hand, some interesting variants that are tested to show the robustness in efficiency and efficacy of Brotli's approach and, on the other hand, some open problems that need further attention from the research community and are thus stated for future investigation.

Section 4 will detail how Brotli encodes the LZ77-phrases identified by the first phase above. This section includes most of the software engineering tricks that allow Brotli to achieve a succinct encoding of those phrases over arbitrary file types and short file lengths, such as the ones indicated with (i)–(iv) above. We will aim at keeping this discussion as much as possible at the algorithmic level, thus avoiding any specific technicalities (for which

[5] http://caniuse.com/#search=brotli

we refer the reader to the Github repository and to RFC 7932 [1]), while still offering enough details to allow us to comment on the impact of the various compression steps, experimented in Subsection 4.1.

Section 5 will offer a thorough, systematic experimental evaluation of Brotli's performance and its comparison with several other compressors that are "optimal" in some scientifically accurate meaning of the term (LZOpt [12], Booster [11]); that are the most well-known members of some data compression family (i.e. Gzip, Bzip2, PPMD [21, 27], xz [7, 20]); that are the state-of-the-art in offering high decompression speed and very good compression ratio (i.e. LZHAM [14], LZFSE [16], Zopfli [26], ZStd [6]); or that offer the highest decompression speeds (i.e. LZ4 [5], Snappy [17]). These experiments will be performed over three well-known sets of files of various kinds and lengths, thus providing a thorough picture of Brotli's performance against state-of-the-art compressors in various settings.[6] We believe that these figures, apart from being meaningful per se, can inspire the research upon new data-compression problems as stated in the concluding Section 6.

2 AN OVERVIEW OF BROTLI

The encoder library of Brotli offers 12 *quality levels* (from 0 to 11). They are compression modes that trade compression speed for compression effectiveness: higher quality levels are slower but yield better compression ratios. Our discussion will concentrate on the last quality level, 11, where all of Brotli's algorithmic novelties are deployed.

A Brotli compressed file is composed of a collection of so-called *meta-blocks*. Each meta-block holds up to 16 MiB and is composed of two parts: a *data part*, which stores the LZ77-compressed input block, and a *header*, which contains information needed to decode the data part. The compression of each block follows the classic LZ77-compression scheme and consists of two main phases:

Phase I: Compute a good entropic LZ77-parsing – The goal of this phase is to compute an LZ77-parsing that ensures good *entropic* properties about the distribution of the features describing its constituting LZ77-phrases. These properties will be exploited in Phase II via a *specific collection* of Huffman codes, whose structure and algorithmic properties will be detailed next, in order to achieve a dense compressed output for that meta-block.

Every LZ77-phrase describes a substring of the meta-block by means of a *copy part* and a *literal part*. The *copy part* can refer either to a substring previously occurring in the sliding window (possibly a previous meta-block) or to an entry in a static dictionary that has been fixed in advance and thus is not transmitted with the compressed block. The *literal part* consists of a sequence of literals explicitly stored in the compressed stream. Such substrings are encoded in a sophisticated way through Brotli *commands*, here simplified to highlight the most important algorithmic issues (for details see RFC 7932 [1]). A Brotli-command consists of a quadruplet:

(*copy length*, *copy distance*, *literals length*, *sequence of literals*)

The integers *copy length* and *literals length* encode the length of the copy part and the length of the literal part, respectively. The *copy distance* refers either to the distance of the previous occurrence of the same substring in the sliding window (possibly a previous meta-block) or to an entry in the static dictionary. The static dictionary

[6] For other types of benchmarks, please refer to: https://quixdb.github.io/squash-benchmark/.

is composed of a set of "base words", i.e., strings of different lengths, along with a set of transformations that can be applied to such words. Each base word and the transformation applied to it are encoded in the copy distance via a proper encoding scheme detailed in the RFC above. Brotli distinguishes between the two cases (copy from before or copy from the dictionary) as follows: if the copy distance is less than some value, defined in terms of the *sliding window* and the starting position of the current command in the text, then the copy part points to a previous substring of length *copy length* and distance *copy distance*; otherwise the copy part is an entry in the static dictionary.

Phase II: Compute a succinct encoding of LZ-phrases – Every component of a Brotli command is compressed by means of a properly designed Huffman code (see the RFC [1] for details, or [25, 27] for classic references on the subject). However, instead of using *just one* Huffman code to compress all the values in each command, Brotli defines a sophisticated compressed format that employs a variable number of Huffman codes to achieve a greater compression ratio. This choice is motivated by the fact that the distribution of the frequency of a symbol is strongly influenced by its *kind* (whether it is a length, a distance, or a literal), its *context* (i.e., its neighboring symbols), and its *position in the text*. Briefly (since we will give further details in the next sections), each meta-block stores in the header the collection of Huffman codes that are used to compress the commands contained in its data part. The set of "symbols" occurring in the fields of the commands are logically organized into three different and independently handled *streams*: namely length, distance, and literal. Each stream is further partitioned into *blocks*; finally, blocks are *clustered* and each cluster of blocks is assigned a numerical, distinct *block id*. Both the *partitioning* and the *clustering* are performed by taking into account the entropic properties of the underlying set of symbols occurring in the clustered blocks. For example, if the files to be compressed have mixed content (such as HTML pages, JSON snippets, etc.), then block boundaries might mark the beginning and the end of the different homogeneous parts, and clustering could group together file parts that express homogeneous content spread all around these files.

For lengths, all values laying in blocks with the same *block id* are encoded using the same Huffman code. For distances and literals, the choice of the Huffman code to use does not depend solely on the *block id* but also on their *context*, so each *cluster* of blocks employs not just one Huffman code but *a set* of Huffman codes of canonical type, whose preamble is further compressed as detailed in Section 3.5 of [1]. Section 4 better illustrates these algorithmic steps.

As a last optimization Brotli exploits the fact that in many kinds of texts, such as natural language or tabular data, there are substrings that are equal except for small differences such as substitution, insertion, or deletion of few characters. Brotli then deploys *relative (copy) distances* which express the *delta* between the current copy-distance and the *previous* copy-distance in the compressed stream. In this way a run of slightly different distances in the compressed stream is changed into a run of *relative* distances, which should take less compressed space.

The choice of this sophisticated encoding format makes the goal of Phases I and II ambitious, namely to compute the LZ77-parsing of the input meta-block that admits a block partitioning and clustering that yields the lowest Huffman-compressed size for the resulting streams of symbols originated from the LZ77-phrases.

In the next sections we illustrate the algorithms adopted by the implementation of Brotli, which is open-source and available to the public at GitHub (see footnote 3), to address the challenges posed by both phases, along with a thorough experimental evaluation of their time/space effectiveness. In doing so, we will concentrate on *a series of algorithmic problems* that arose from the implementation of Brotli and we will relate them to what is known in the literature in order to characterize their novelties, special features, and future challenges posed to algorithm researchers. Therefore, this paper is not to be read only as the description of an "efficient and efficacious compressor" but also, in our opinion, as the collection of new algorithmic and engineering problems in the data compression and string-matching realms. In particular, in Section 3, we will prove that the LZ77-parsing problem dealt in Brotli in its Phase I is computationally difficult and then highlight some new algorithmic challenges whose solutions could impact beneficially onto the design of Brotli itself. In addition, we will compare Brotli's algorithms with the *bicriteria data compressor*, recently introduced in [10], in order to highlight their similarities and, more importantly, their differences. This will allow us to identify new interesting venues for algorithmic research that deserve much attention from the scientific community. Phase II is, instead, much newer in its design and thus we will discuss it from an algorithm-engineering point of view with the goal of describing Brotli's solutions and their challenges, which are left open for future research. All our algorithmic discussions are enriched by a thorough experimental analysis of Brotli's modules, with the intention of supporting our theoretical considerations on engineering and performance grounds, which should make it evident that algorithmic contributions in this setting could induce significant contributions onto Brotli's performance in practice.

3 PHASE I: COMPUTE A GOOD ENTROPIC LZ77-PARSING

The classic definition of LZ77-parsing of a string $\mathcal{S}$, having length n and built over an alphabet $\Sigma = [\sigma]$, is as follows [27]: it consists of a sequence of phrases $p_1, \ldots, p_k$ such that phrase p_i is either a single character of Σ (encoded as $\langle 0, c \rangle$) or a substring that occurs in the prefix $p_1 \cdots p_{i-1}$, and thus it can be copied from there. Once the LZ77-parsing has been identified, each phrase is represented via pairs of integers $\langle d, \ell \rangle$, where d is the distance from the (previous) position where the copied phrase occurs, and ℓ is its length.

The main insight used in the most efficient LZ77-parsing strategies to date (see e.g. [9, 10, 13, 22]) is to represent every parsing of $\mathcal{S}$ as a path over a weighted graph $\mathcal{G}$ in which each vertex v_i corresponds to position i in $\mathcal{S}$, and each edge (v_i, v_j) corresponds to a possible LZ77-phrase, whose cost is given by the number of bits needed to encode it according to a chosen compressed scheme (to be detailed next). This way, the most succinct LZ77-parsing of $\mathcal{S}$ corresponds to the Single-Source Shortest Path (SSSP, for short) from v_1 to v_n over $\mathcal{G}$. Since $\mathcal{G}$ is a DAG, a SSSP can be computed in time linear in the number of graph edges. Unfortunately, however, $\mathcal{G}$ might have $\Theta(n^2)$ edges, such as when $\mathcal{S} = a^n$. Therefore, a central aspect related to the efficiency of such methods is identifying asymptotically-smaller subgraphs of $\mathcal{G}$ that retain an optimal SSSP. An algorithmic cornerstone in efficiently solving the SSSP problem on such subgraphs relies on designing an efficient *Forward Star Generation* (FSG) strategy, that is, an algorithm that generates such subgraphs on-the-fly along with their SSSP. This is an inevitable requirement because it would take too much time to *materialize* the entire $\mathcal{G}$ and then *prune* it.

The known algorithms for graph-pruning and FSG rely on the properties of the (class of) encoders used to compress the LZ77-phrases, since they determine the edge weights. For the simplest case of encoders based on fixed-length codewords, each edge has the same weight

and so the SSSP-computation boils down to the problem of finding a path with a minimal number of edges. It is very well known that the greedy FSG strategy, which selects the edge representing the longest copy at each vertex, yields an optimal solution: so, in this case, $\mathcal{G}$ can be pruned to a graph with just $O(n)$ edges. However, a fixed-length codeword encoder needs $2 \log n$ bits per LZ77-phrase, which results in a very poor compression ratio. For the more interesting case of *universal integer coders*, Ferragina *et al.* [13] proposed a pruning strategy (hereafter denoted by Fast-FSG) that generates a small subgraph of $\mathcal{G}$ of $O(n \log n)$ edges, taking $O(1)$ amortized time per edge and $O(n)$ words of space. That subgraph was proved to include an optimal SSSP.

In this paper we take one step further and investigate, for the first time in the literature, the complexity of the *Minimum-Entropy LZ77-Parsing Problem*, namely, the problem of finding the most succinct LZ77-parsing in which *entropic* statistical encoders (such as Huffman or Arithmetic) are used to compress the LZ77-phrases. In the language of SSSP and graph $\mathcal{G}$, the goal is to compute the SSSP of a weighted graph $\mathcal{G}$ in which *entropic* statistical encoders are used to define the edge weights. The change in the time complexity of the problem with respect to the other formulations is not negligible as will be apparent soon.

DEFINITION 1. *The* minimum-entropy LZ77 parsing problem *(MELZ, for short) asks for a LZ77-parsing π of an input string $\mathcal{S}$ that achieves the minimum bit encoding when copy-lengths and copy-distances of the LZ77-phrases in π are compressed using two (possibly different) entropy coders.*

Formally speaking, let $\mathsf{dists}(\pi)$ *(resp.* $\mathsf{lens}(\pi)$*) denote the sequence of copy-distances d (resp. copy-lengths ℓ), taken from all LZ77-phrases $\langle d, \ell \rangle$ in π, where d may be* 0*, in which case ℓ denotes a singleton character. The problem MELZ asks for a parsing π that minimizes the function:* $s^\star(\pi) = |\pi| \cdot (H_0(\mathsf{dists}(\pi)) + H_0(\mathsf{lens}(\pi)))$.

This is arguably the most interesting version of the LZ77-parsing problem discussed to date in the literature, because entropic encoders consistently yield the most succinct compressed files in practice, and indeed they are the choice in most industrial-grade compressors, dating back to `gzip`. Unfortunately, the techniques developed for the universal coding case [10, 13] cannot be readily extended to handle this case, as the cost in bits of an LZ77-phrase (and thus the weight of an edge in $\mathcal{G}$) depends on the empirical distribution of the distance- and length-components of all other LZ77-phrases already selected for the current parsing, so they cannot be fixed in advance when instantiating the graph $\mathcal{G}$. This makes the MELZ problem very difficult to solve intuitively and in fact we prove in Appendix A that a significant variant of it is $\mathcal{NP}$-Hard. Nevertheless, Brotli provides an efficient heuristic for finding an *entropically-good* parsing π, as detailed in the next section.

3.1 The parsing strategy adopted by Brotli

Given the computational difficulty of devising an efficient and optimal minimum-entropy parsing strategy, Brotli deploys a sophisticated heuristic inspired by the theoretical foundations laid down by the optimal LZ77-parsing previously outlined. As such, parsing is recast as a shortest path problem on a weighted DAG $\mathcal{G}$ with the addition of some approximations that are employed to achieve an efficient, practical algorithm that deals with the eventual use of Huffman codes. The main technical challenges tackled by Brotli are thus (i) devising a suitable strategy that assigns an *entropic cost* to edges in $\mathcal{G}$ (see below for a definition), (ii) defining a suitable sub-graph $\widetilde{\mathcal{G}}$ of $\mathcal{G}$ that includes a close approximation of the optimal

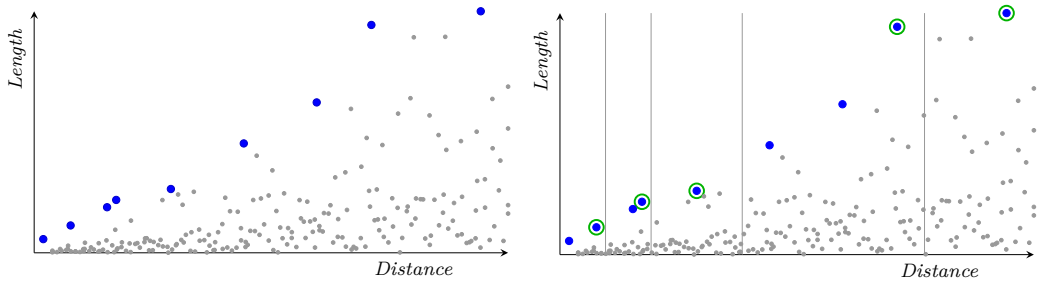
SSSP, and (iii) designing an efficient FSG algorithm for generating on-the-fly $\widetilde{\mathcal{G}}$ along with the computation of its SSSP.

Assigning "entropic" weights to $\mathcal{G}$'s edges. As mentioned earlier, it is not possible to precisely assign bit costs to edges in $\mathcal{G}$ because the cost of a LZ77-phrase depends on the empirical distributions of copy-distance and copy-length components in the parsing. Therefore, the approach taken by Brotli is to approximate these weights through a sequence of *cost models* c_1, c_2, ... where each c_{i+1} assigns to the LZ77-phrase $\langle d, \ell \rangle$ a cost $c_{i+1}(d, \ell)$ that depends on the result of the SSSP computation over the graph $\mathcal{G}$ weighted with costs c_i. More specifically, the first cost model c_1 assigns to each edge a constant cost; while the subsequent cost models c_{i+1} are defined by first computing a shortest path $\pi_i^\star$ on $\mathcal{G}$ weighted according to the previous cost model c_i and then assigning to the edge corresponding to the LZ77-phrase $\langle d, \ell \rangle$ the entropic cost $c_{i+1}(d, \ell) = \log\left(\frac{n}{\#\mathsf{Dists}(d, \pi_i^\star)}\right) + \log\left(\frac{n}{\#\mathsf{Lens}(\ell, \pi_i^\star)}\right)$. This process could stop when the difference between the costs of the last two shortest paths falls below a fixed threshold: the last path provides the LZ77-parsing returned by Phase I. In practice Brotli chooses to stop after just two iterations, because our systematic tests over a large (proprietary) dataset showed that this gave the best trading between SSSP encoding and time efficiency.

Defining a suitable subgraph $\widetilde{\mathcal{G}}$. The *pruning strategies* proposed in the literature are based on some specific properties of fixed-length codes or universal codes [13] that do not hold for the statistical codes we wish to deal with. Brotli devises an effective pruning strategy that hinges upon the following simple observation: short copies do typically occur close to the current position to be matched, whereas long copies are typically found farther in the input text. This implies that small distances have higher frequencies than long distances, but the latter typically refer to longer copies than the former. In terms of space occupancy, this means that far&long copies are able to amortize the bit cost incurred by long distances because they squeeze long phrases. Brotli tries to reinforce this dichotomy (close&short copies versus far&long copies) at each iteration of the SSSP computation by only considering *dominating edges*, namely edges for which there is no other edge outgoing from the same vertex that corresponds to a longer, or equally long, and closer copy. Conversely, we say that the latter edges are *dominated* by the former one.

Handling runs of literals. From Section 2 we know that a Brotli-command is composed of a LZ77-like copy component followed by a sequence of literals, while our definition of $\mathcal{G}$ involves only LZ77-like copy phrases. A natural and conceptually simple way to handle this is to change the definition of $\mathcal{G}$ by extending the edges in $\mathcal{G}$ in order to include literal runs of different lengths. However, this must be done cautiously because the resulting graph $\mathcal{G}$ would eventually acquire $\Theta(n^2)$ edges. To tackle this Brotli defines two classes of edges: one for copies and one for literal runs. Any path in $\mathcal{G}$ will be composed of a mix of copy and literal edges, which will eventually be fused and represented as single Brotli-commands in the encoding phase.

Literal edges are handled differently from copy edges. Let us assume that the shortest path computation has already found the shortest paths from v_1 to vertexes v_2, ..., v_{i-1} and must process vertex v_i: while for copy edges we update the cost of successive vertexes v_{j_1}, v_{j_2}, ... by generating the *forward star* of v_i, namely (v_i, v_{j_1}), (v_i, v_{j_2}), ...; for the literal edges we update the cost of v_i by considering the *backward star* of v_i, namely (v_1, v_i), (v_2, v_i), ..., (v_{i-1}, v_i). This could be time consuming, so Brotli reduces the number of incoming edges to just one edge



(a) All dominated copies (b) Copies selected by Fast-FSG (circled).

Fig. 1. An illustration of dominating copies. Each substring copy is depicted as a point in the Cartesian space with distances on the x-axis and lengths on the y-axis. In the plots, dominated copies are gray while dominating copies are blue. Figure (a) shows all dominated/dominating copies, while Figure (b) highlights in green the dominating copies selected by Fast-FSG. Recall that Fast-FSG selects one dominating copy per "window" of text preceding the currently examined position. In the picture, copies are illustrated for increasing distance from the current position, these windows are nested and extend from the y-axis to each vertical bar, with $\alpha = 2$.

through the following observation. Let $\mathsf{LitCost}(k, l)$ be the cost of representing the substring $\mathcal{B}[k, l-1]$ as a literal run in a Brotli command, and let $c^\star(i)$ be the cost of the shortest path from v_1 to v_i. The cost of the shortest path from v_1 to v_i with a literal run as the last edge is given by $\min_{j=1}^{i-1}\{c^\star(j) + \mathsf{LitCost}(j+1, i)\} = \min_{j=1}^{i-1}\{c^\star(j) + \mathsf{LitCost}(1, i) - \mathsf{LitCost}(1, j)\} = \mathsf{LitCost}(1, i) + \min_{j=1}^{i-1}\{c^\star(j) - \mathsf{LitCost}(1, j)\}$. This search can be accelerated to $O(1)$ time by keeping the vertex v_j that minimizes $c^\star(j) - \mathsf{LitCost}(1, j)$ among the vertexes v_j that precede v_i in $\mathcal{G}$ via a proper priority queue data structure. Actually, Brotli employs a *queue* of positions, sorted according to the expression illustrated above, in order to take care of the possibility to relatively-encode the distance of the back references (see Section 4).

An efficient FSG algorithm. The final algorithmic challenge of Phase I is to compute efficiently the dominating edges outgoing from each vertex in the pruned graph $\widetilde{\mathcal{G}}$ (see Figure 1a). Brotli is based upon a strategy that we denote here as Treap-FSG, and for simplicity of description we assume that the meta-block coincides with the compression window in which LZ77-copies can be searched for. It processes the input meta-block $\mathcal{B}$ via a single left-to-right scan driven by an iterator i such that a *dynamic Treap* $\mathcal{T}$ [23] indexes the pairs $\langle S_j, j\rangle$ where $S_j = \mathcal{B}[j, n]$ is the j-th suffix of the meta-block and $j < i$. We remind the reader that a Treap is a binary tree in which the keys (here, suffixes of $\mathcal{B}$) are organized as in a binary search tree via their lexicographic order, whereas the priorities (here, positions of the suffixes) are organized as in a max-heap; this way, the closer the starting position of a suffix S_j is to the currently processed meta-block's position i, the closer the pair $\langle S_j, j\rangle$ is to the heap's root.

Figure 2 shows an example of a static Treap built over the entire string $\mathcal{B} = ananas$. The Treap is defined recursively on the set of six pairs $\{\langle ananas, 1\rangle, \langle nanas, 2\rangle, \langle anas, 3\rangle, \langle nas, 4\rangle, \langle as, 5\rangle, \langle s, 6\rangle\}$. The root is labeled with the pair $\langle s, 6\rangle$ which is the one in the set having the maximum priority. This selection splits the set into two subsets: the pairs corresponding to suffixes that are lexicographically smaller than s, and the pairs corresponding to suffixes that are lexicographically larger than s. The latter subset is empty in our example, so that the right child of the $\mathcal{T}$'s root is NULL; all the other pairs will be then used to build recursively the Treap forming the left child of the root.

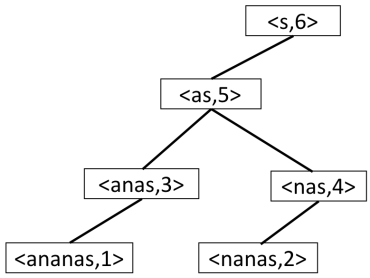


Fig. 2. A Treap built on the string $\mathcal{B} = ananas$, positions are counted from 1. Every node contains the pair $\langle S_j, j\rangle$ where S_j is the j-th suffix of $\mathcal{B}$.

Let us now go back to the description of the algorithm Treap-FSG. At a generic step, it processes some suffix S_i by searching it into the current Treap $\mathcal{T}$ (recall that it contains suffixes S_j starting at previous positions $j < i$). Let $S_{i_1}, \ldots, S_{i_m}$ be the sequence of suffixes examined by the search for S_i in $\mathcal{T}$. The algorithm first determines the set of LZ77-phrases $\{\langle d_{i_j}, \ell_{i_j}\rangle\}_{j=1}^{m}$ corresponding to these suffixes via a comparison between S_i against each suffix S_{i_j} and setting $d_{i_j} = i - i_j$ and $\ell_{i_j} = \mathsf{Lcp}(S_i, S_{i_j})$; secondly, suffix S_i is inserted into $\mathcal{T}$ as a pair $\langle S_i, i\rangle$ and brought to its root (because the Treap contains pairs $\langle S_j, j\rangle$ with $j < i$) via proper tree rotations (as detailed in [23]); finally, the set of dominating copies $M_i^{\mathcal{T}}$ is extracted from $\{\langle d_{i_j}, \ell_{i_j}\rangle\}_{j=1}^{m}$ and returned to the shortest path computation.

Property 1. *Algorithm Treap-FSG returns all dominating copies starting at each position i in $\mathcal{B}[1, n]$.*

Proof. Let $M_i^{\star}$ be the set of *all* dominating copies starting from position i of $\mathcal{B}$. A search in $\mathcal{T}$ works by percolating the binary tree and thus restricting the lexicographically-sorted range suffixes $[L, R]$ such that $L < \mathcal{B}[i, n] < R$. Recall that those suffixes start before position i since they have been previously inserted in the Treap. Initially, we can assume that L is the empty string, and R is the infinite string consisting of repeating the largest possible character. Since $\mathcal{T}$ is a binary search tree over the lexicographic order of its indexed suffixes, the search at a node x updates L with the suffix at this node if the comparison leads to follow the right child, otherwise it updates R. In order to drive the lexicographic search, the algorithm computes the Lcp between the searched suffix $\mathcal{B}[i, n]$ and the suffix at node x.

Now, in order to prove the statement above, we have to consider the two possible cases (we will interchangeably talk about nodes in $\mathcal{T}$ and suffixes stored into them).

Case $x \in M_i^{\mathcal{T}} \implies x \in M_i^{\star}$. Let us assume, by contradiction, that $x \in M_i^{\mathcal{T}}$ but $x \notin M_i^{\star}$, so x has been returned by Treap-FSG and there is a longer copy $z \in M_i^{\star}$ that dominates x. This means that z denotes a suffix which starts after x's suffix in $\mathcal{B}$ and shares a longer Lcp with $\mathcal{B}[i, n]$. Since we assumed that $x \in M_i^{\mathcal{T}}$, then x is returned by Treap-FSG but the node z is not returned. The latter implies that the search has not encountered z during the search for $\mathcal{B}[i, n]$, even if z should appear higher in the Treap because it occurs after x in $\mathcal{B}$ (remember the max-heap property on the suffixes' positions). Therefore there must exist an ancestor $y \in \mathcal{T}$ of x that makes the search for $\mathcal{B}[i, n]$ diverge from z and lead to x. Assuming that $z < y < x$ holds (the other case is symmetric and not treated here), then the lexicographic comparison between y and $\mathcal{B}[i, n]$ cannot lead to the right (i.e. towards x) because $\mathsf{Lcp}(z, \mathcal{B}[i, n]) > \mathsf{Lcp}(x, \mathcal{B}[i, n])$

by the dominating hypothesis above. Hence it should go to the left (i.e. towards z) thus leading to the contradiction.

Case $x \in M_i^\star \implies x \in M_i^{\mathcal{T}}$**.** Since x is a dominating suffix, all suffixes that occur after x in $\mathcal{B}$ (and before i) share an Lcp with $\mathcal{B}[i,n]$ shorter than $\ell = \mathsf{Lcp}(\mathcal{B}[i,n], x)$. Our goal is to show that the search for $\mathcal{B}[i,n]$ in $\mathcal{T}$ must meet node x, and thus must insert it into $M_i^{\mathcal{T}}$. Let us consider the root-to-x path π in $\mathcal{T}$: where $\pi = y_1\, y_2 \cdots y_k\, x$. Any node $y_j \in \pi$ is a suffix occurring after x in $\mathcal{B}$ (because of the max-heap property on the suffixes' position) and sharing an $\mathsf{Lcp}\big(y_j, \mathcal{B}[i,n]\big) < \ell$ (because of the above dominating property of x). Therefore, when the search for $\mathcal{B}[i,n]$ is in y_j, it must continue on the child y_{j+1} because $\mathsf{Lcp}\big(y_j, \mathcal{B}[i,n]\big) < \ell = \mathsf{Lcp}(\mathcal{B}[i,n], x)$ and x descends from y_{j+1}.

□

The time efficiency of Treap-FSG depends on the number of nodes traversed during the search for $\mathcal{B}[i,n]$, since the Lcp-computation can be performed in $O(1)$ time through the proper use of a Suffix Array built on $\mathcal{B}$ and a Range-Minimum Query data structure built on top of the Lcp array [3]. To this end, we can adopt the analysis of Allen and Munro [2] taking care of the fact that the Treap $\mathcal{T}$ can be looked at as a sort of self-adjusting tree with insertion driven by a move-to-the-root heuristic, as shown in the following property.

PROPERTY 2. *Algorithm Treap-FSG requires* $\Omega(n/\log_\sigma^2 n)$ *node visits per examined position in* $\mathcal{B}$*, where* $n = |\mathcal{B}|$.

PROOF. Let $i_1, i_2, \ldots, i_m$ be a permutation of the first $m = \frac{n}{\log_\sigma n}$ integers that takes $\Omega(m^2)$ time to be inserted in a tree with a move-to-the-root balancing strategy. Let $\perp$ and $\top$ be, respectively, the smallest and greatest symbols in Σ. Finally, let s_j be the textual representation in base $\sigma - 1$ of the integer i_j, padded with enough $\perp$ on the left if shorter than $1 + \lfloor \log_{\sigma-1} m \rfloor$. Notice that the textual representation reflects the ordering of integers: i.e. $s_j < s_k$ iff $i_j < i_k$.

We construct the text $\mathcal{B} = \top s_1 \top s_2 \cdots \top s_m$ and process it via the algorithm Treap-FSG. The insertion of all suffixes starting at symbols $\top$ requires the visit of $\Omega(m^2)$ nodes in the Treap, because of the definition of sequence $i_1, \ldots, i_m$ and the observation above on the ordering correspondence between integers i_j and strings s_j. Given that $\mathcal{B}$ has length $m(1 + \log_{\sigma-1} m) = O(m \log_\sigma m) = \Theta(n)$, then the average work per position is $\Omega(m^2/n) = \Omega(n/\log_\sigma^2 n)$ and so this is the worst-case too. □

The worst-case complexity of Treap-FSG is thus not very appealing. It could be observed that the self-adjusting strategy has indeed a much better *amortized* complexity of $\mathcal{O}(\log n)$ [24] but, unfortunately, this argument cannot be applied here because the "splay" operation used by Treap-FSG would break the heap property of the Treap. Nevertheless, Brotli avoids all these problems with an implementation of the FSG-strategy, called hereafter Brotli-FSG, which adds a few effective heuristics on top of Treap-FSG that are based on *constants* derived from a systematic experimental analysis over a large (proprietary) corpus of variegate file types (see also RFC 7932 [1] for details):

Top Pruning of the Treap: Since meta-blocks of few MBs, as the ones managed by Brotli, contain most probably every possible sequence of four characters, it is convenient to distinguish between short copies and long copies. More precisely, Brotli-FSG handles "short&close" copies with length $\ell \leq 4$ and distance $d < 64$ through char-by-char comparison, while it manages copies of length $\ell > 4$ by using a *forest* of Treaps, each one containing suffixes that start with the same 4-chars prefix. The rationale is that

small copies are convenient in terms of space compression only when their distance is small and their search is sped up by a direct char-by-char comparison instead of a Treap navigation.

Bottom Pruning of the Treap: Each Treap in the forest discards nodes that are at distance greater than 64 from the root. Due to the max-heap property, this implies discarding suffixes that are farthest away from the currently examined position. As a result, some of these discarded suffixes could be dominating copies, and they would be lost by Brotli-FSG (more on this issue next).

Skipping Suffixes: When a copy of length $\ell > 325$ is found, the suffixes starting in the next $\ell - 64$ positions are neither searched nor inserted into the Treap. The rationale behind this is that the parsing almost surely would select this long copy in the shortest path computation, and so the edges starting in the next positions would not be used. However, the suffixes starting in the last 64 characters covered by the copy are anyway inserted into the Treap because they could induce dominating copies for later positions in the text of not negligible length.

Stopping Condition and Copy Elision: When a copy of length $\ell > 128$ is found in a Treap, the search is stopped and the node corresponding to that copy is removed from the Treap, whereas the currently searched suffix is inserted into that node. The rationale is that, even if the discarded suffix could originate a longer Lcp in a subsequent search with respect to what can be obtained with $\mathcal{B}[i, n]$, the latter provides in any case a copy of at least 128 chars, which is considered "good enough". This strategy makes it possible to trade better running times for a (likely) small degradation in the succinctness of the LZ77-copies.

Notice that all these tweaks imply that the algorithm might not find all dominating copies for each position (in fact, it cannot find more than 64 copies per position, given the pruning at depth 64 of $\mathcal{T}$) and that it might output some dominated copies in lieu of the dominating ones. The next section will investigate experimentally the impact of these optimizations on the quality of the copies returned with the navigation of the Treap.

3.2 Experimental results

All the experiments presented in this section, and in the rest of the paper, will be performed over three files of different types and 1GiB in size each. These files have been built by extracting one block of that size, starting at a random position, within three classic datasets:

- Census: U.S. demographic information in tabular format (type: database);
- Mingw: `mingw` software distribution (type: mix of source codes and binaries)[7];
- Wikipedia: dump of English Wikipedia (type: natural language).

We note that our experiments only have the intention of commenting on the properties and the efficiency of the algorithmic choices made in the design of Brotli over some classic and publicly available datasets. It goes without saying that Brotli is an open-source general-purpose data compressor, so it can be used on any file type.

Since it is impractical to compute the set of all dominating copies for all positions in $\mathcal{B}$, we compare the set of dominating copies found by Brotli-FSG with those found by the algorithm Fast-FSG proposed in [9, 10, 13]. This latter algorithm samples the set of dominating copies in a very specific way: specifically, for each position i of which we want to compute the

[7]Thanks to Matt Mahoney – http://mattmahoney.net/dc/mingw.html.

dominating copies, it considers a set of K nested windows $W_j = [i - w_j, i - 1]$ where the nesting property is satisfied by requiring that $w_j = \alpha w_{j-1}$ for some integral value $\alpha \geq 2$; then Fast-FSG finds, for each window W_j, the longest copies whose starting position falls in W_j. Figure 1(b) provides an illustration of the dominating copies selected by Fast-FSG.

The comparison between the dominating copies found by Brotli-FSG and Fast-FSG consists of comparing the frequencies of the selected LZ77-phrases taking windows of 4MiB for Brotli-FSG, and 22 nested windows of maximum size 4MiB for Fast-FSG (hence $w_1 = 2, \alpha = 2$). For ease of reading, these frequencies are illustrated by means of *heatmaps* in Figure 3–4, that is, a Cartesian space where copy distances are mapped on the x-axis, copy lengths are mapped on the y-axis, and each 2D-point is assigned a color that depends on the frequency of the corresponding LZ77-phrase given by these two coordinates. However, some care has to be taken in order to make the visualization illustrative and meaningful.

Firstly, notice that, if there is a phrase $\langle d, \ell \rangle$ starting at position $\mathcal{B}[i]$, then the phrases $\langle d, \ell - 1 \rangle$, $\langle d, \ell - 2 \rangle, \ldots,$ *and* $\langle d, 1 \rangle$ are also present in $\mathcal{B}$ starting at positions $\mathcal{B}[i + 1]$, $\mathcal{B}[i + 2]$, $\ldots$, $\mathcal{B}[i + \ell - 1]$. So, if $\langle d, \ell \rangle$ has been found by Brotli-FSG, then most of these (nested and shorter) phrases are also going to be generated; conversely, if $\langle d, \ell \rangle$ has *not* been generated, then probably their nested copies are also not generated by Brotli-FSG. The same argument applies to Fast-FSG. As a result we have that, if Brotli-FSG generates a long copy of length ℓ that Fast-FSG does not generate, then this difference is going to be reported $\sim \ell$ times in the difference of the heatmaps, which is inconvenient because it makes it difficult to understand whether a set of differences has been yielded by a long copy with many nested copies or by many copies scattered around $\mathcal{B}$. To overcome this issue, we filter the list of LZ77-phrases returned by both strategies to include only the longest (leftmost) run of LZ77-phrases starting in consecutive positions and with the same distance.

Secondly, since two distinct distances that are "close" in $\mathcal{B}$ should have *on average* comparable frequency, it makes sense to consider them equivalent for this study. Hence, we group the phrases filtered above into *buckets*, and consider that two phrases $\langle d, \ell \rangle$ and $\langle d', \ell' \rangle$ belong to the same bucket if $\ell = \ell'$ and $\lfloor \log_2 d \rfloor = \lfloor \log_2 d' \rfloor$. For each bucket we compute the number of copies that fall into it.

The plots are composed of squares, one distance-bucket long and one single copy-length tall. Each square is assigned a color that encodes the difference of their frequencies: shades of red indicate higher frequencies in that bucket of LZ77-phrases in Fast-FSG, while colors towards blue-violet indicate higher frequencies in that bucket of LZ77-phrases in Brotli-FSG.

We concentrate hereafter on Census (see Figure 3), because the main conclusions we discuss below can be drawn also from the plots of the other two datasets in Figure 4: Mingw and Wikipedia. Not surprisingly, we notice that Fast-FSG has many more copies of length up to four (shades of red in the left part of the plots) because of the "Top-Pruning" tweak adopted by Brotli, which implies that no copy of length up to 4 and distance greater than 64 can ever be found by Brotli-FSG. The other main differences are mostly grouped in four clusters: two colored blue/purple (favouring Brotli-FSG) and two colored red/orange (favouring Fast-FSG). Specifically, we have that Brotli-FSG finds more copies in the blue ellipsoid (distances up to $4,096$, lengths starting from 16) and in the green ellipsoid (distances starting from $2,048$, lengths up to 16); while Fast-FSG finds more copies in the grey ellipsoid (distances less than 32 and copies longer than 4) and in the red ellipsoid (both far distances and long lengths). The differences in the blue ellipsoid can be explained by noticing that Fast-FSG finds only one copy per window, while Brotli-FSG does not have this limitation. This gives a substantial advantage to Brotli-FSG, since finding more copies with small/medium distance and longer

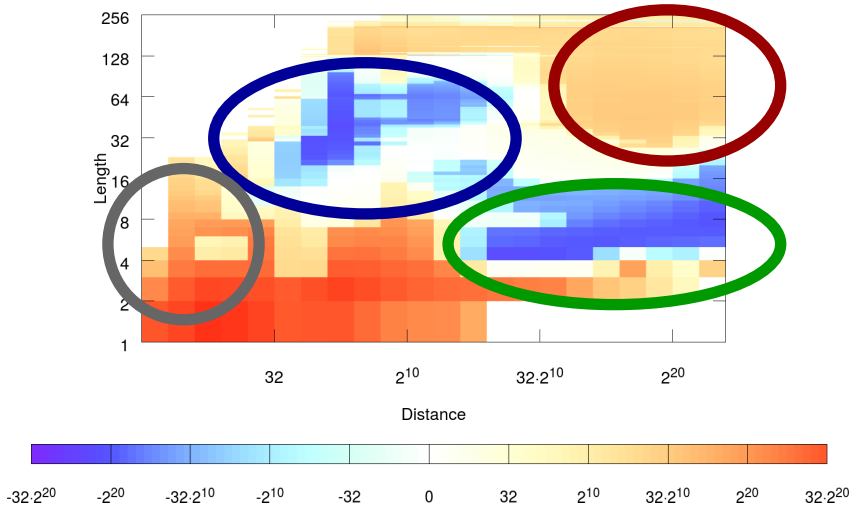
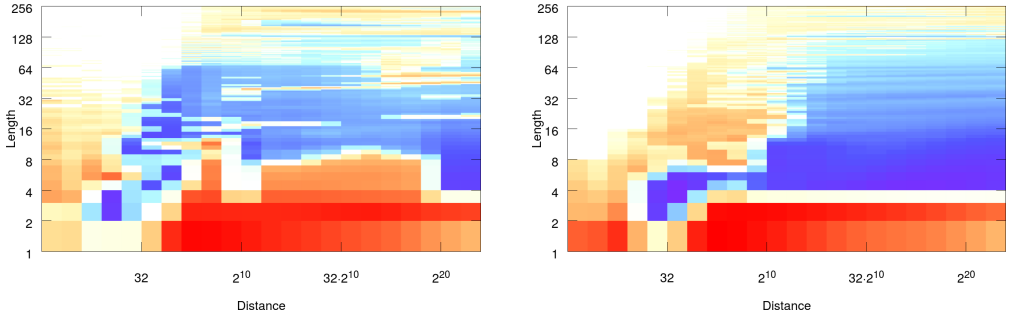


Fig. 3. Differential heatmap of Fast-FSG minus Brotli-FSG over the dataset Census (with the four quadrant highlighted with four ellipsoids, see comments in the text). Shades of red signals more copies for Fast-FSG, while shades of blue/purple signal more copies for Brotli.

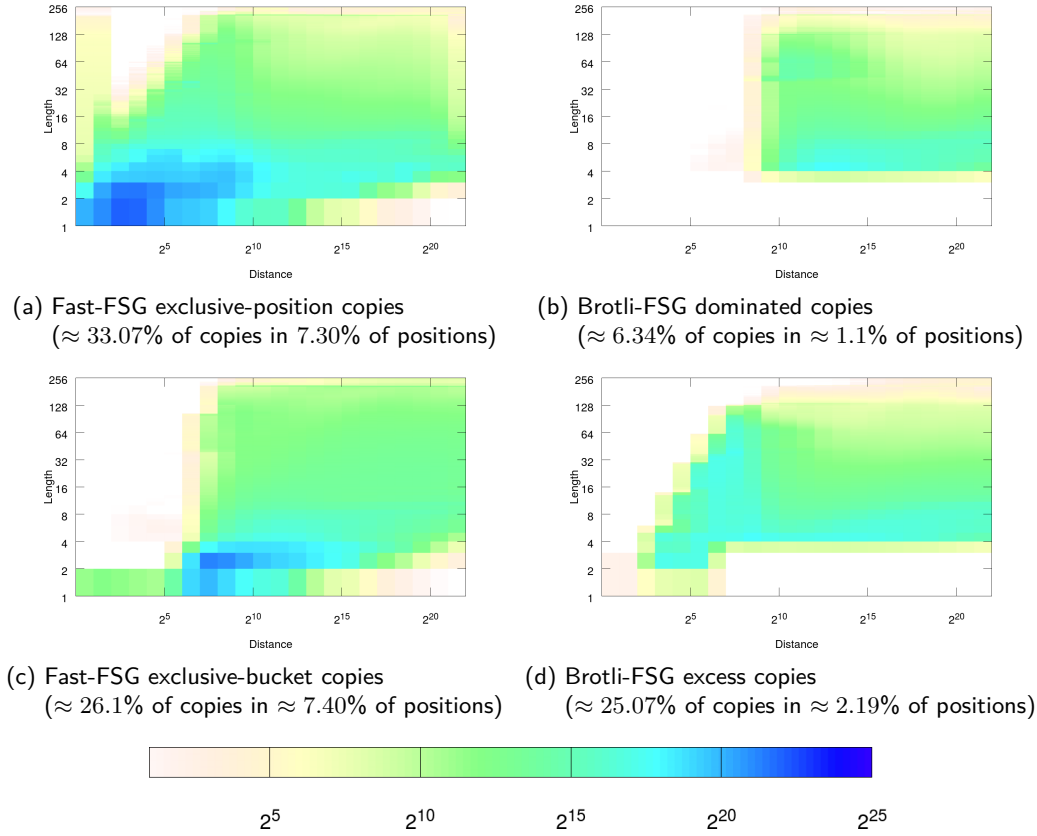


(a) Mingw (b) Wikipedia

Fig. 4. Differential heatmap of Fast-FSG minus Brotli-FSG over the two datasets Mingw and Wikipedia. The colours follow the grading shown in Figure 3.

length can help the iterative shortest path computation pick up more copy patterns and improve the empirical entropic cost of the parsing. The differences in the green ellipsoid might have two explanations: either these additional copies are found because, as we just mentioned, Fast-FSG is restricted to finding only one copy per window, or because these copies are dominated by other (longer) copies, thus not generated by Fast-FSG. Notice that Brotli-FSG might generate a dominated copy because the dominating ones could have been "skipped" due to a previous long copy. The "skipping heuristic" can also explain the presence of the grey ellipsoid. The red ellipsoid (favoring Fast-FSG) can be explained by the "pruning heuristic" applied by Brotli-FSG, which removes farther suffixes from the Treap.

Starting from these hypothesis, we delved deeper in the analysis and distinguished four types of copies, which are illustrated in Figure 5 over the dataset Census.



(a) Fast-FSG exclusive-position copies
($\approx$ 33.07% of copies in 7.30% of positions)

(b) Brotli-FSG dominated copies
($\approx$ 6.34% of copies in $\approx$ 1.1% of positions)

(c) Fast-FSG exclusive-bucket copies
($\approx$ 26.1% of copies in $\approx$ 7.40% of positions)

(d) Brotli-FSG excess copies
($\approx$ 25.07% of copies in $\approx$ 2.19% of positions)

Fig. 5. Detailed analysis for Census, with an explanation of the color scale in the bar above.

Fast-FSG exclusive-position copies: This measures the number of copies found by Fast-FSG in positions that have been skipped by Brotli-FSG. First of all, we have counted the number of positions that are skipped by Brotli-FSG, which varies greatly with respect to the dataset: 72.2% for Census, 38.1% for Mingw and 12.1% for Wikipedia. Then, looking at the heatmap illustrated in Figure 5(a) we notice that Fast-FSG exclusive-position copies are probably not selected by the shortest path algorithm (and thus by Brotli-FSG) because they are "covered" by a longer copy that induced the skipping of that position. Overall, these copies take about 33% of the total number of copies on Census (this percentage is 43% on Mingw and 11.5% on Wikipedia). The consequence is that the number of skipped positions is not negligible and arguably has a significant impact in speeding up the computation, making Brotli fast indeed.

Brotli-FSG dominated copies: This measures the number of dominated (and thus sub-optimal) copies computed by Brotli-FSG because of the "skipping" and "copy elision" strategies (as defined above). These are copies found by Brotli-FSG which are shorter and farther than copies found by Fast-FSG for a given position. Looking at the heatmap in Figure 5(b), we notice that the set of Brotli-FSG-dominated copies is low (i.e. about 6%), and thus it has an arguably small impact in the final compression ratio achieved by Brotli. Moreover, they are distributed mostly uniformly at distances larger than 256

and more. This seems to suggest that the vast majority of these dominated copies are the byproduct of the "skipping heuristic" that avoids the insertion of suffixes into the Treap, since the minimum distance at which this can happen is 64.

Fast-FSG exclusive-bucket copies: This measures the number of dominating copies that Brotli-FSG loses due to pruning, skipping, and copy elision. With respect to Figure 1(b), these are the circled copies that are found by Fast-FSG but not found by Brotli-FSG. Looking at the heatmap in Figure 5(c), we note that the number of times Fast-FSG finds longer copies for a given distance-bucket (overall positions) is significant and equal to 26% of the total copies (this figure is 39% for Mingw and 62% for Wikipedia), but they are concentrated in less interesting parts of the plot: namely, for lengths up to 4 and distances longer than $1,024$. These copies are short and far, indeed. Therefore, they arguably do not reduce Brotli-FSG's decompression efficiency and compression ratio with respect to Fast-FSG.

Brotli-FSG excess copies: This measures the ability of the Treap to find more than one copy for the same distance bucket. With respect to Figure 1(b), these are the uncircled blue dominating copies that are found by Brotli-FSG and not by Fast-FSG. The heatmap in Figure 5(d) confirms the conclusions derived from the differential heatmap in Figures 3 that Brotli-FSG finds a greater number of copies in the "upper/middle-left" quadrant of the plot: this is 25% of the total copies for Census (this figure is 19% for Mingw and 11.8% for Wikipedia). These are *useful* copies because they are close to the current position and have large copy-length.

Overall plots (a)–(d) show that for a substantial number of positions of the input meta-block the two strategies actually return the same set of copies. In fact, the number of positions where the copies found by the two approaches differ can be bounded above by about 18%, simply by adding the percentages of positions indicated in the captions of the various sub-pictures of Figure 5. Since Fast-FSG is proved to find the dominating copies, Brotli-FSG's heuristic is also guaranteed to find most of them. Therefore, we can conclude that the engineering tweaks introduced by Brotli's Treap strategy do not hinder too much the final compression performance. As far as the differences between the two strategies are concerned, it can be inferred that Brotli-FSG is very effective at small distances, but it seems to lose a share of interesting dominating copies occurring farther in the text. This seems to suggest that a "hybrid" strategy mixing elements of the Treap with elements of the suffix array-based strategy adopted in Fast-FSG might yield more effective results.

To test this hypothesis, we evaluated the compression ratio and decompression speed obtained when using either Brotli-FSG, or Fast-FSG, or a "hybrid" strategy that combines Brotli-FSG and Fast-FSG, over the three datasets with chunks of 4MiB (Table 1).[8]
In particular, this is the list of FSG strategies evaluated:

- Brotli-FSG: the Brotli-FSG strategy instantiated with a window of length equal to the chunk size;
- Fast-FSG: the Fast-FSG strategy instantiated with $w_1 = 4$, $\alpha = 2$ and 20 nested levels;
- Brotli-FSG + Fast-FSG: a strategy that emits, for each position of $\mathcal{B}$, the *union* of the matches returned for that position by Brotli-FSG and Fast-FSG;

[8]We also repeated the experiments on chunks of 16MiB, and noticed, unsurprisingly, a slight improvement in compression ratio at the cost of a slight decrease in decompression speed. We do not report them for ease of reading.

Table 1. Compression performance with different FSG strategies. We detail: compression ratio (ratio between compressed size and dataset length, in %) and compression / decompression speed and the difference in compressed size (Δ) with respect to Brotli with all optimizations enabled. Each dataset is split in chunks of 4MiB, which are compressed individually.

Dataset	FSG strategy	C. Ratio (%)	Dec. Speed (MiB/sec)
Census	Brotli-FSG	3.51	512.86
	Fast-FSG	3.46	525.43
	Brotli-FSG + Fast-FSG	3.46	524.22
	Brotli-FSG$(1, 512)$ + Fast-FSG$(512, 4M)$	3.49	521.58
	Brotli-FSG$(1, 32K)$ + Fast-FSG$(32K, 4M)$	3.52	516.37
Mingw	Brotli-FSG	25.09	203.67
	Fast-FSG	25.05	204.67
	Brotli-FSG + Fast-FSG	25.05	205.03
	Brotli-FSG$(1, 512)$ + Fast-FSG$(512, 4M)$	25.07	204.26
	Brotli-FSG$(1, 32K)$ + Fast-FSG$(32K, 4M)$	25.10	203.78
Wikipedia	Brotli-FSG	18.87	214.14
	Fast-FSG	18.87	214.49
	Brotli-FSG + Fast-FSG	18.87	215.46
	Brotli-FSG$(1, 512)$ + Fast-FSG$(512, 4M)$	18.87	214.73
	Brotli-FSG$(1, 32K)$ + Fast-FSG$(32K, 4M)$	18.94	212.75

- Brotli-FSG$(1, 512)$ + Fast-FSG$(512, 4M)$: a strategy that returns, for each position of $\mathcal{B}$, all the matches with distance smaller than 512 provided by Brotli-FSG and, conversely, matches longer than 512 provided by Fast-FSG for that position;
- Brotli-FSG$(1, 32K)$ + Fast-FSG$(32K, 4M)$: as before, but splitting at distance 32KiB instead of 512.

From the figures we notice that the differences among the various strategies are quite modest, being less than 1.4% with respect to Brotli-FSG on all datasets and chunk sizes. This is not surprising if we consider that the differences in the set of returned matches between Brotli-FSG and Fast-FSG are concentrated in few positions, as observed above.

As far as time efficiency is concerned, Table 2 illustrates a break-down of the running times of Brotli's compression on chunks on 4MiB when using either the Brotli-FSG or the Fast-FSG strategies. In particular, we report the overall time taken by the FSG step, the (iterative) shortest path computation and all the optimizing steps performed in Phase II (Section 4). We first observe that, even though Brotli-FSG has worse theoretical bounds than Fast-FSG (namely, $\mathcal{O}\left(n^2/\log n\right)$ vs $\mathcal{O}(n \log n)$, Property 2), it is much more efficient in practice, being from 6 to 13 times faster than Fast-FSG: the Fast-FSG strategy needs to compute the Suffix Array of the meta-block, taking about 3% of its total running time (hence about 50% of Brotli-FSG), and about 4.8% for each of its 20 levels (one per nested window, see the beginning of this section). The Fast-FSG might be accelerated by reducing the number of levels and through a software engineering effort aimed at making the computation of one level more time-efficient. However, since on our datasets Brotli-FSG takes less than 12.5% of

the total compression time, we think that any improvement in this respect would have a limited impact and thus suggest that any further efforts in improving the time efficiency of Brotli compression be directed toward the other steps of the compression algorithm. An obvious candidate is the shortest path computation, which takes around 52–81% of the total compression time when using the Brotli-FSG strategy. These figures inspire some interesting research directions that we report in Section 6.

Table 2. Running time breakdown of the various compression steps over chunks of 4MiB. For each step we report, in parenthesis, its contribution to the total compression time (reported in the last column).

Dataset	FSG strategy	FSG		Shortest Path		Phase II		C. Time
		secs	(%)	secs	(%)	secs	(%)	secs
Census	Brotli-FSG	234	(9.6%)	1,979	(81.5%)	216	(8.9%)	2,430
	Fast-FSG	3,201	(59.7%)	1,952	(36.4%)	207	(3.9%)	5,361
Mingw	Brotli-FSG	559	(9.7%)	3,011	(52.1%)	2,212	(38.3%)	5,783
	Fast-FSG	3,904	(43.0%)	3,021	(33.3%)	2,157	(23.8%)	9,084
Wikipedia	Brotli-FSG	653	(12.5%)	3,419	(65.6%)	1,142	(21.9%)	5,216
	Fast-FSG	4,234	(48.3%)	3,420	(39.0%)	1,111	(12.7%)	8,766

4 PHASE II: COMPUTE A SUCCINCT ENCODING OF LZ-PHRASES

Brotli employs a variety of new techniques to improve the efficacy of compressing the LZ77-phrases selected in Phase I. We will concentrate on the most important ones, in a sequence that recalls the way they are applied to compress the LZ77-phrases, then we will analyze their impact on the overall compression performance by executing a large set of experiments over the three datasets: Census, Mingw, Wikipedia. For more details we refer the reader to [1].

Static dictionary: the LZ77 dictionary is expanded with a *static dictionary*, about 100Kib in size, which is embedded in the decompressor and thus does not have to be encoded in the output compressed file. Multiple ad hoc methods have been used to generate a set of candidate words; the ranking and final selection was done by calculating how many bits of saving each word alone brought to a mixed (proprietary) corpus of small files. At the end, each dictionary word has 121 different forms, given by applying a *word transformation* to a base word in the dictionary. These transformations are fully described in [1], as well as the static dictionary, and they are much too complicated to be reported here. Entries in the static dictionary are encoded via *escape values* represented by means of distances larger than the maximum backward copy distance at which a LZ77-phrase can reference its copy (hence they are outside the compression window). In summary, any reference to the dictionary is represented as a triplet that specifies the length of the copy (between 4 and 24), the index of the dictionary word (according to the previously mentioned escape values), and the index of the word transformation to be applied.

Relative pointers: Distance codes ranging from 0 to 15 are interpreted as sort-of-*deltas* with respect to the distance of the previous phrase in the parsing: if d_{prev} is the distance of the previous phrase in the compressed stream, then a distance code of 0 is interpreted as distance d_{prev}; a distance code of 1 is interpreted as $d_{\text{prev}} + 1$, 2 as $d_{\text{prev}} - 1$, etc. Distance codes d from 16 onward are interpreted as $d - 16$.

Blocks partitioning: Let $S = \sigma_1, \sigma_2, \ldots$ be a sequence of symbols in a meta-block to be encoded. Recall that symbols can be of three types: *literals*, *copy/literal lengths*, or *copy distances*; hence they can be any value of a component of the quadruples generated by Brotli as LZ77-phrases. Block partitioning divides S into blocks and assigns to each block a distinct Huffman code that will be used for encoding its symbols. The partitioning is obtained via an iterative optimization loop that is executed three times, once for each symbol type, and eventually assigns to each block an `id` that denotes the *block type*. The idea behind the iterative optimization loop is very similar to the iterative computation of the SSSP in Phase I. The main routine takes a set of Huffman codes as input and then implements a dynamic programming strategy that finds the optimal block partitioning for S under the assumption that S's symbols must be coded using these Huffman codes. Clearly, this generally yields a non-optimal solution because the input Huffman codes might not be the ones derived from the best partitioning. Therefore, the algorithm works in rounds: each round computes the optimal block partitioning against a given set of Huffman codes, then updates those Huffman codes, then another round is executed, for a total of 10 rounds, thus balancing compression effectiveness and compression speed. The initial Huffman codes are built over a distribution derived from a few chunks sampled from S.

Blocks clustering: The optimization loop above does not account for the cost of serializing the Huffman trees: the amount of compressed space taken out by storing these trees might not be negligible if the number of "similar" blocks and the cardinality of their alphabets are not negligible either. To account for this, Brotli joins *blocks* (not block types) with similar Huffman codes in order to reduce their number. That is, it (i) computes the optimal Huffman code for each block; (ii) clusters different blocks that have "similar" Huffman trees (see below); (iii) computes a new representative Huffman code for each cluster by computing the optimal Huffman code on the *union* of their blocks; (iv) assigns a `cluster_id` to each cluster; and (v) turns this `cluster_id` into the `block_id` of its clustered blocks.

Technically speaking, the clustering algorithm takes as input a set of N Huffman codes and produces in output a (smaller) set of M Huffman codes by keeping a priority queue Q including pairs of Huffman codes, sorted by the "space saving" induced on the sequence S if the blocks compressed by the two codes were merged and one unique Huffman code were generated from the merged blocks. At the beginning, Q is initialized with all possible pairs of Huffman codes; then the clustering algorithm proceeds *greedily* by merging the best pair of codes that are stored at the root of Q: as a result, this new merged Huffman code is inserted in Q, all pairs referring to the two originating Huffman codes are removed from Q, and new pairs referring to the new merged code are inserted in Q. Since the algorithm has a time/space complexity quadratic in N, the clustering works first on chunks of 64 Huffman codes (by considering all of their 2016 pairs), and finally it is applied to the Huffman codes derived by those chunks. The clustering stops when the improvement is not smaller than a given ratio (first merging phase) or the number of clusters is not greater than 256 (second merging phase). A few other optimizations are eventually applied and not described here. We refer the reader to the code on GitHub and to the description in the RFC [1].

Encoder switching: There are two ways in which Brotli signals which Huffman code must be used at any time: through *explicit switching*, using *block switches* dispersed into the compressed stream, and through *contextual switching*, as detailed below.

(1) *Explicit*: A triple $(\mathsf{Type}, \mathsf{id}, \ell)$ is stored in the compressed stream, called *block switch*, to inform the decoder that the next ℓ symbols of kind Type (e.g. Type = Literal or Type = CopyDistance) belong to a block with *block_id* id. For example, a block switch $(\mathsf{Literal}, 3, 5)$ signals to the decoder that the next five literals belong to a block with block_id 3. Notice that many different blocks can be tagged with the same block_id, because the same Huffman code can be reused for different blocks. The map between block_id and the Huffman tree to be used for these symbols is encoded in a context map (see point 2 below). All the Huffman trees to be used for encoding/decoding blocks are stored at the beginning of the compressed meta-block.

(2) *Contextual*: For copy distances and literals (but not for copy lengths, which use a single Huffman tree), the choice of the Huffman tree to encode the symbol σ_i depends on its *context*. Symbols of different kinds define "context" differently: for literals, the context is given by its two preceding characters $\sigma_{i-2}\ \sigma_{i-1}$, while for a copy distance its context is given by the copy length of the same phrase (i.e., ℓ). A *context map* is encoded at the beginning of the compressed meta-block and it gives the Huffman tree to be used for encoding σ_i given its block_id and its context_id. In particular the context_id is generated via a proper non-injective mapping onto the co-domain $[0, 63]$ whose technical details are provided in Section 7.1 of the RFC [1].

4.1 Experimental results

In this section we study the impact of the previously described compression techniques by augmenting Brotli's open-source reference with a set of *suppression switches*:

- NoDict: disables the static dictionary;
- NoRel: disables the encoding via relative pointers;
- NoPart(Dst), NoPart(Len) and NoPart(Lit): disables block partitioning for copy distances, copy lengths and literals, respectively; put differently, under this setting, there will be only one block type for the selected symbols;
- NoContext(Dst) and NoContext(Lit): disables contextual compression for distances and lengths, respectively.

We ran the experiments by first splitting each one of our three dataset into chunks of 4MiB and then compressing each chunk many times, each with a different configuration of the above switches. We performed every compression with quality level 11 (the highest). We computed compressed sizes as the sum of the compressed sizes of the chunks; we computed compression/decompression times in the same manner as well. We chose chunks of 4MiB because we wish to consider a *realistic* scenario in which data has to be accessed randomly and thus the chunk length is a common choice trading long, highly compressible chunks and short, fast-accessible chunks. A complete picture with other chunk lengths, ranging from 256KiB to 1GiB, and other configurations of the above switches can be found at http://pages.di.unipi.it/farruggia/dcb/ and we now briefly comment on them.

We carried out our experiments with two different machines: a virtualized 12-cores AMD Opteron 6238 operating at 2,600MHz and 128GiB of DDR3 memory at 1,600MHz, used for compression, and a "bare-metal" 6-cores Intel Xeon X5670 CPUs clocked at 2,933MHz and with 32GiB of DDR3 memory at 1,333MHz, used for collecting decompression statistics. Both machines run Ubuntu 14.04. We chose this dual-machine setting because compression usually requires more time and memory than decompression, so we use a (virtualized) more powerful machine to accelerate compression and a (bare-metal) less powerful machine to obtain more accurate timings in decompression. This must be taken into account when

interpreting the compression/decompression timings that cannot be, hence, compared in absolute terms because of the two different experimental settings involved. We will always refer to *in-memory* operations when evaluating compression/decompression speeds. An in-memory compressor first fully loads the data in memory, then encodes/decodes it in memory, and finally writes the result on disk; this way, the timing of the compression/decompression step excludes any costly read/write from the disk. In order to fairly and accurately measure times, we implemented a front-end compressor with a standardized CLI interface for every compressor evaluated in this paper, using the publicly available APIs[9]. We compiled every compressor with the same compiler (`gcc 4.8.4`) and with the same flags `-O3 -NDEBUG -march=native`). Decompression times are the arithmetic mean of 10 decompressions, with a cache wiping after each decompression.[10]

The following tables report experimental results according to the following notation: compression ratio is computed as the ratio between compressed size and dataset length (in %), compression/decompression speeds are measured as commented above, and the value between parenthesis indicated with "(Δ)" reports the relative difference in percentage with respect to the same value computed by Brotli with all optimizations enabled. We note that in the following tables the higher the compression/decompression speeds are the better (hence, positive Δ), while in the case of compression ratios we have an improvement for smaller values of that ratio, so that we use the positive sign for Δ when it achieves smaller ratios.

Baseline comparison. Here we compare Brotli with no optimization against Brotli with all optimization turned on.

Dataset	Configuration	Ratio (Δ)		Compr. Speed MiB/sec (Δ)		Decompr. Speed MiB/sec (Δ)	
Census	Full	3.51	(0.0%)	0.29	(0.0%)	298.01	(0.0%)
	NoOptimizations	4.09	(-16.5%)	0.62	(113.3%)	327.33	(9.8%)
Mingw	Full	25.04	(0.0%)	0.11	(0.0%)	94.18	(0.0%)
	NoOptimizations	28.24	(-12.8%)	0.39	(249.3%)	113.07	(20.1%)
Wikipedia	Full	18.59	(0.0%)	0.14	(0.0%)	116.51	(0.0%)
	NoOptimizations	20.33	(-9.3%)	0.35	(154.3%)	138.16	(18.6%)

Disabling all optimizations affects the compression ratio by a minimum of about 9.3% (Wiki) to a maximum of about 28.2% (Census). Both compression and decompression speeds are significantly affected by the use of all optimizations: compression is slowed down by a factor of 1.1–2.5, while decompression is slowed down between 10–20%, which actually means a slow down of 20–30MiB/sec.

Static dictionary. Improvements in compression ratio are quite modest across all our datasets when the static dictionary is used: they range from 0% on Census (where the static dictionary is actually used in a very limited way because of the tabular nature of Census) to about 1.5% on Wikipedia. The impact on compression and decompression speeds is modest as well.

[9]With the exception of PPMD, for which there is no publicly available API and therefore has been tested using 7z's implementation.

[10]The implementation of these front ends is available on GitHub: https://github.com/farruggia/random_decs. All experimental data produced for this paper is collected and illustrated in a companion website reachable at the address http://pages.di.unipi.it/farruggia/dcb.

Dataset	Configuration	Ratio (Δ)		Compr. Speed MiB/sec (Δ)		Decompr. Speed MiB/sec (Δ)	
Census	Full	3.51	(0.0%)	0.29	(0.0%)	298.01	(0.0%)
	NoDict	3.51	(0%)	0.30	(2.7%)	297.95	(-0.0%)
Mingw	Full	25.04	(0.0%)	0.11	(0.0%)	94.18	(0.0%)
	NoDict	25.08	(-0.2%)	0.12	(3.3%)	94.72	(0.6%)
Wikipedia	Full	18.59	(0.0%)	0.14	(0.0%)	116.51	(0.0%)
	NoDict	18.87	(-1.5%)	0.15	(5.4%)	119.70	(2.7%)

However, it must be said that the static dictionary proves its effectiveness only over small files, which occur frequently in the Web context scenario (pages, posts, etc.). Therefore, we ran another experiment in which we split the same datasets into chunks of 64KiB. The results below show that the static dictionary significantly affects the compression ratio on small files that are well structured (e.g. XML) and with natural language content as it occurs in Wikipedia, although this comes at the cost of a (almost equivalent) slow down in decompression speed.

Dataset	Configuration	Ratio (Δ)		Compr. Speed MiB/sec (Δ)		Decompr. Speed MiB/sec (Δ)	
Census	Full	4.01	(0.0%)	0.13	(0.0%)	427.00	(0.0%)
	NoDict	4.01	(0%)	0.13	(4.7%)	426.76	(-0.1%)
Mingw	Full	30.74	(0.0%)	0.08	(0.0%)	171.17	(0.0%)
	NoDict	31.04	(-1.0%)	0.08	(0%)	175.45	(2.5%)
Wikipedia	Full	22.80	(0.0%)	0.08	(0.0%)	149.63	(0.0%)
	NoDict	25.15	(-10.3%)	0.08	(0%)	167.39	(11.9%)

Explicit encoder switch. Here we investigate the explicit Huffman Tree switching mechanism for literals (through suppression switch NoPart(Lit)), lengths (NoPart(Len)), distances (NoPart(Dst)) and altogether (NoPart). Improvements in compression ratio range from 1.6% in Wikipedia to 4.5% in Mingw, while the penalty in compression speed , which is about 40–70KiB/sec, is negligible. These improvements mainly come from the literal and length explicit encoder switch, as distance encoder switching does not affect performance much, both in compression ratio and compression/decompression speeds. Disabling explicit encoder switching for all three kinds of symbols has a penalty close to the sum of each individual penalty, suggesting that these optimizations are largely independent.

Dataset	Configuration	Ratio	(Δ)	Compr. Speed MiB/sec	(Δ)	Decompr. Speed MiB/sec	(Δ)
Census	Full	3.51	(0.0%)	0.29	(0.0%)	298.01	(0.0%)
	NoPart	3.63	(-3.4%)	0.33	(12.3%)	306.45	(2.8%)
	NoPart(Lit)	3.57	(-1.6%)	0.30	(4.2%)	300.98	(1.0%)
	NoPart(Len)	3.56	(-1.4%)	0.31	(5.6%)	301.53	(1.2%)
	NoPart(Dst)	3.53	(-0.5%)	0.30	(2.2%)	299.56	(0.5%)
Mingw	Full	25.04	(0.0%)	0.11	(0.0%)	94.18	(0.0%)
	NoPart	26.16	(-4.5%)	0.18	(62.8%)	97.27	(3.3%)
	NoPart(Lit)	25.58	(-2.2%)	0.15	(35.5%)	94.07	(-0.1%)
	NoPart(Len)	25.43	(-1.6%)	0.12	(9.6%)	95.11	(1.0%)
	NoPart(Dst)	25.22	(-0.7%)	0.12	(2.8%)	93.65	(-0.6%)
Wikipedia	Full	18.59	(0.0%)	0.14	(0.0%)	116.51	(0.0%)
	NoPart	18.90	(-1.6%)	0.18	(27.1%)	120.51	(3.4%)
	NoPart(Lit)	18.63	(-0.2%)	0.15	(8.8%)	117.25	(0.6%)
	NoPart(Len)	18.71	(-0.6%)	0.15	(9.4%)	117.49	(0.8%)
	NoPart(Dst)	18.75	(-0.8%)	0.15	(5.4%)	117.66	(1.0%)

Contextual encoder switch. This has a significant impact when applied on literals (around 1.6–4%), while on distances it brings little improvements (less than about 0.2%). In both cases, compression and decompression speeds are not affected much.

Dataset	Configuration	Ratio	(Δ)	Compr. Speed MiB/sec	(Δ)	Decompr. Speed MiB/sec	(Δ)
Census	Full	3.51	(0.0%)	0.29	(0.0%)	298.01	(0.0%)
	NoContext	3.65	(-4.1%)	0.29	(1.4%)	300.98	(1.0%)
	NoContext(Lit)	3.65	(-4.0%)	0.29	(1.4%)	299.65	(0.6%)
	NoContext(Dst)	3.51	(0%)	0.29	(1.3%)	297.61	(-0.1%)
Mingw	Full	25.04	(0.0%)	0.11	(0.0%)	94.18	(0.0%)
	NoContext	25.49	(-1.8%)	0.12	(8.6%)	98.67	(4.8%)
	NoContext(Lit)	25.45	(-1.6%)	0.12	(7.7%)	95.99	(1.9%)
	NoContext(Dst)	25.08	(-0.2%)	0.11	(-0.4%)	94.89	(0.8%)
Wikipedia	Full	18.59	(0.0%)	0.14	(0.0%)	116.51	(0.0%)
	NoContext	19.14	(-2.9%)	0.14	(1.5%)	123.99	(6.4%)
	NoContext(Lit)	19.10	(-2.7%)	0.14	(1.2%)	123.05	(5.6%)
	NoContext(Dst)	18.63	(-0.2%)	0.14	(0.5%)	117.25	(0.6%)

Contextual+Explicit switching. Here we evaluate the impact of disabling encoder switching optimization on both literals and distances.

Dataset	Configuration	Ratio (Δ)		Compr. Speed MiB/sec (Δ)		Decompr. Speed MiB/sec (Δ)	
Census	Full	3.51	(0.0%)	0.29	(0.0%)	298.01	(0.0%)
	NoPart(Lit), NoContext(Lit)	3.69	(-5.2%)	0.30	(5.1%)	315.17	(5.8%)
	NoPart(Dst), NoContext(Dst)	3.56	(-1.4%)	0.31	(7.1%)	300.41	(0.8%)
Mingw	Full	25.04	(0.0%)	0.11	(0.0%)	94.18	(0.0%)
	NoPart(Lit), NoContext(Lit)	26.03	(-4.0%)	0.15	(34.3%)	105.77	(12.3%)
	NoPart(Dst), NoContext(Dst)	25.47	(-1.7%)	0.12	(8.4%)	95.69	(1.6%)
Wikipedia	Full	18.59	(0.0%)	0.14	(0.0%)	116.51	(0.0%)
	NoPart(Lit), NoContext(Lit)	19.24	(-3.5%)	0.15	(8.9%)	126.44	(8.5%)
	NoPart(Dst), NoContext(Dst)	18.74	(-0.8%)	0.15	(9.7%)	118.48	(1.7%)

Switching on literals has an impact on compression efficiency, ranging from a minimum of about 3.5% on Wikipedia to a maximum of about 5.2% on Census. On the web pages dataset (not shown here, see the accompanying website), the impact is even larger (around 7%) when considering longer blocks up to 16Mib. On the other hand, switching on distances seems less significant, with a gain on compression ratio less than 2% on all datasets.

Switching consistently slows down compression by about 10–40KiB/sec. In decompression, switching has a limited impact when done on distances, while on literals the impact varies greatly across datasets, ranging from $\sim$ 17MiB/sec over 315MiB/sec on Census (5.8%) to about 21MiB/sec over 105MiB/sec on Mingw (12.3%).

Relative pointers. The impact of using relative pointers is very significant:

Dataset	Configuration	Ratio (Δ)		Compr. Speed MiB/sec (Δ)		Decompr. Speed MiB/sec (Δ)	
Census	Full	3.51	(0.0%)	0.29	(0.0%)	298.01	(0.0%)
	NoRel	3.84	(-9.4%)	0.50	(71.6%)	303.48	(1.8%)
Mingw	Full	25.04	(0.0%)	0.11	(0.0%)	94.18	(0.0%)
	NoRel	26.41	(-5.5%)	0.15	(31.2%)	93.13	(-1.1%)
Wikipedia	Full	18.59	(0.0%)	0.14	(0.0%)	116.51	(0.0%)
	NoRel	19.08	(-2.6%)	0.21	(50.0%)	117.94	(1.2%)

Compression ratio improvements range from a minimum of $\sim$ 2.6% on Wikipedia to a maximum of $\sim$ 9.4% on Census. Relative pointers improve compression whenever there are copies in the input text with low edit-distance: while a standard LZ77-parsing needs to begin a new phrase whenever there is an edit event (insertion, deletion, or substitution), Brotli only encodes the number of inserted or deleted characters in the currently parsed phrase. This is why the impact is the highest on Census, in which CSV-entries are very close syntactically.

The penalty on compression speed is significant as well: it ranges from about 31.2% on Mingw (i.e. a difference of about 40KiB/sec) to a maximum of about 71.6% on Census (i.e. a difference of about 210KiB/sec). Actually, this penalty is at times even higher, such as 238%, over a dataset composed of web pages and chunks of size up to 16 Mbs, as illustrated on the aforementioned accompanying website containing the whole set of experiments.

On the other hand, the penalty on decompression speed is modest, topping at about 1.8% on Census or about 4MiB/sec over $\sim$ 300MiB/sec in absolute terms.

Wrapping up. Brotli exhibits excellent compression ratios mainly by virtue of a compact representation and two important optimizations:

(1) *Relative pointers*, which succinctly encode copies with low edit distance;
(2) *Literal encoder switching*, which aims at emulating *prediction by partial matching* [19, 27] in the compression framework laid out by Brotli.

The results are briefly summarized in this table:

Option enabled	Compression ratio improvement	Compression speed slowdown	Decompression speed slowdown
Static dictionary	$< 1.5\%$	$<$ 10KiB/sec	$<$ 4MiB/sec
(chunks 64KiB)	$< 10.3\%$	$<$ few KiB/sec	$<$ 18MiB/sec
Relative pointers	$2.6 - 9.4\%$	$40 - 210$KiB/sec	$<$ 4MiB/sec
Explicit switch	$1.6 - 4.5\%$	$40 - 70$KiB/sec	$<$ 7MiB/sec
Context switch	$1.8 - 4.1\%$	$<$ 10KiB/sec	$<$ 7MiB/sec
Literal switch	$3.5 - 5.2\%$	$10 - 40$KiB/sec	$10 - 17$MiB/sec
Distance switch	$0.8 - 1.7\%$	$10 - 20$KiB/sec	$1 - 2$ MiB/sec

5 COMPARISON WITH STATE-OF-THE-ART

In the final experiment, we use the same files and the same experimental setting of the previous sections to compare Brotli against several other compressors that are either "optimal" in some scientifically accurate meaning (LZOpt, Booster), or are the most well-known members of a data compression family (i.e. Gzip, Bzip2, PPMD), or are the state-of-the-art for high decompression speed and very good compression ratio (i.e. LZHAM, LZFSE, Zopfli, xz, ZStd), or are the state-of-the-art for the highest decompression speeds (i.e. LZ4, Snappy).

- Bzip2, Gzip: *Boost* implementation of these two well-known compressors. Source code available at www.boost.org/doc/libs/1_62_0/libs/iostreams/.
- Booster: is the reference implementation of the *Compression Booster* technique proposed in [11] based on the BW-Transform and aimed at turning a number of base 0-th order entropy compressors into a k-th order entropy compressor. Precisely, here we tested *Range Encoding* (yielding a compression format somewhat similar to Bzip2) and *Huffman coding*. Source code available at http://people.unipmn.it/manzini/boosting/.
- LZFSE: Apple's implementation of the LZ77-strategy using the Finite State Entropy (FSE) encoder inspired by the *asymmetric numeral systems* of [8] for squeezing LZ77-phrases. Source code available at https://github.com/lzfse/lzfse.
- LZHAM: a LZ77-compressor engineered for high compression ratios at (relatively) high decompression speeds. Source code available at https://github.com/richgel999/lzham_codec_devel.
- LZOpt: is the efficient implementation of the Bit-Optimal LZ77-strategy proposed by [10, 13]. Source code available at ttps://github.com/farruggia/bc-zip.
- LZ4: a LZ77 compressor with excellent decompression speed. Source code available at https://github.com/lz4/lz4.
- PPMD: an implementation of the PPM-compressor, which achieves among the highest compression efficacies. Source code available at http://www.7-zip.org.

Table 3. Detailed performance of the most significant subset of the compressors illustrated in Figure 6. For each compressor we detail: compression ratio (between compressed size and dataset length, in %), compression / decompression speed and the difference in compressed size (Δ) with respect to Brotli with all optimizations enabled. Each dataset is split in chunks of 4MiB, which are compressed individually.

Dataset	Algorithm	Configuration	Ratio %	(Δ) (%)	Comp. Speed MiB/sec	(Δ) (%)	Dec. Speed MiB/sec	(Δ) (%)
Census	Brotli	Full	3.51	(0)	0.27	(0)	633.10	(0)
		NoOptimizations	4.09	(16)	0.56	(105)	730.28	(15)
	LZOpt	TNibble	4.65	(32)	0.79	(193)	1394.22	(120)
		VByte-Fast	5.18	(48)	1.05	(288)	1571.34	(148)
	LZ4	—	6.05	(72)	6.14	(2164)	3051.51	(382)
	LZHAM	Slow	3.52	(0)	0.18	(-34)	448.69	(-29)
		Fast	3.56	(1)	0.18	(-34)	590.15	(-7)
	ZStd	—	3.65	(4)	0.17	(-38)	1339.48	(112)
Mingw	Brotli	Full	25.04	(0)	0.11	(0)	221.94	(0)
		NoOptimizations	28.24	(13)	0.35	(228)	271.13	(22)
	LZOpt	TNibble	31.61	(26)	0.67	(523)	505.33	(128)
		VByte-Fast	34.65	(38)	0.80	(653)	741.68	(234)
	LZ4	—	37.60	(50)	10.25	(9500)	1532.13	(590)
	LZHAM	Slow	25.53	(2)	0.18	(64)	171.02	(-23)
		Fast	25.69	(3)	0.18	(64)	193.57	(-13)
	ZStd	—	26.45	(6)	0.74	(590)	433.81	(95)
Wikipedia	Brotli	Full	18.59	(0)	0.13	(0)	229.38	(0)
		NoOptimizations	20.33	(9)	0.31	(139)	271.23	(18)
	LZOpt	TNibble	22.61	(22)	0.65	(404)	444.32	(94)
		VByte-Fast	26.18	(41)	0.79	(512)	649.97	(183)
	LZ4	—	30.30	(63)	12.24	(9350)	1280.10	(458)
	LZHAM	Slow	19.62	(6)	0.15	(19)	175.66	(-23)
		Fast	19.73	(6)	0.15	(19)	204.12	(-11)
	ZStd	—	19.84	(7)	0.78	(505)	361.73	(58)

- Snappy: a LZ77-compressor with high compression and decompression speeds. Source code available at https://github.com/google/snappy.
- xz: an implementation of the `LZMA/LZMA2` compression algorithms. Source code available at https://tukaani.org/xz/.
- Zopfli: a pseudo-optimal LZ77-compressor that targets DEFLATE's specification. Source code available at https://github.com/google/zopfli.
- ZStd: another LZ77-compressor that deploys the FSE encoder, like LZFSE, but targeting a different trade-off. Source code available at https://github.com/facebook/zstd.

Table 3 and Figure 6 report only the most important findings drawn from comparing Brotli against all those other compressors. For a full picture we refer the interested reader to the (already mentioned) accompanying website.[11]

By looking at the pictorial representation of the trade-off between compression ratio and decompression speed shown in Figure 6, we notice that the Pareto frontier is consistently composed of Brotli at the leftmost part of the compression ratio spectrum, followed by ZStd, then LZOpt and, at the rightmost part of the spectrum, by LZ4, which takes the spot thanks to its incredibly fast decompression speed (by design). On linguistic files (such as Wikipedia), PPMD takes a spot in the leftmost part of the Pareto frontier, but at the cost of an extremely slow decompression speed that makes it an unsuitable candidate for most applicative scenarios.

By looking at the numbers reported in Table 3, we draw the following conclusions:

- Brotli is more succinct than LZOpt: compression ratio gap ranges from 8% to 30% when using TNibble as integer encoder, and from 21% to 42% when using VByte-Fast. On the other hand, LZOpt is faster than Brotli by a factor in the range of 1.4–2.7 with TNibble and 1.6–3 with VByte-Fast.
- Brotli is more succinct than ZStd up to about 7%, but ZStd is about twice as fast as Brotli. However it must be underlined that Brotli uses smaller windows (by default), its compression density and compression speed to a given density tend to be better, it degrades less when many parallel connections are used, and actually the speed difference wrt ZStd starts to have a meaning only when the transfer or loading speeds exceed 1 Gbps.
- Brotli is much more succinct than LZ4, offering compression ratios that are as low as half of those offered by LZ4. On the other hand, LZ4 exhibits decompression speeds on the order of GiB/sec, an order of magnitude higher than what Brotli can offer.

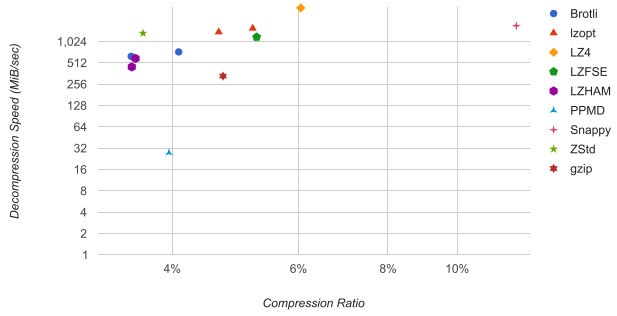
Another interesting compressor is LZHAM: it is both quite succinct and offers "medium" decompression speeds. However its compressed space and decompression time are never part of the Pareto-optimal frontier. In fact, it offers trade-offs very similar to Brotli but it is overall slightly less succinct and slightly slower in decompression: the gap in compression ratio is up to about 6%, and the gap in decompression speed is up to about 30%.

6 CONCLUSIONS

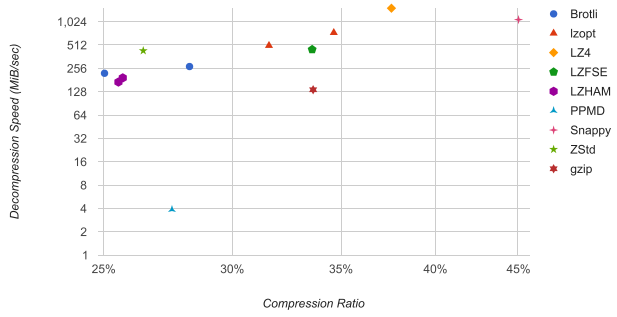
After the birth of Brotli in 2013, there was a revamped interest among the industrial players in the design of general-purpose data compressors. In the previous section we commented on and experimented with Apple's LZFSE and Facebook's ZStd. We hope that our paper, all these new compression tools, and the challenges posed by the processing, transmission, and storage of Big Data will boost the interest of the scientific community in the algorithmic issues underlying `Brotli` and those other compressors. We believe that much study has yet to be done and many results are still to come that could offer interesting performance improvements to those tools and their deploying platforms.

This paper not only offers the first thorough, systematic description and analysis (both algorithmic and experimental) of Brotli's main compression steps, but also raises a few interesting questions that deserve, in our opinion, some attention from the scientific community. The following is a short and partial list of them:

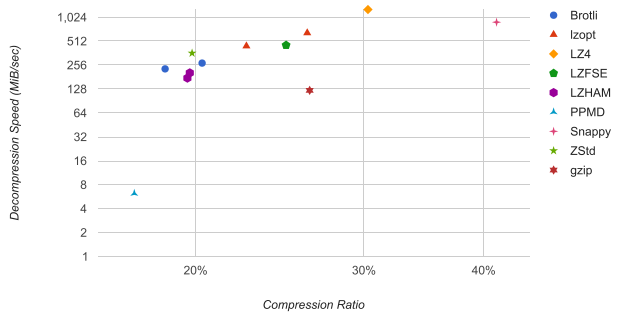
[11] See http://pages.di.unipi.it/farruggia/dcb.



(a) Census



(b) Mingw



(c) Wikipedia

Fig. 6. Graphical depiction of the compression ratio/decompression speed for all compressors we tested in our experiments.

- The shortest path computation executed in Phase I (see Section 3.2) takes about 52–81% of the total compression time when using the Brotli-FSG strategy. A challenge would be to speed it up and/or provide a principled analysis of the strategy implemented in Brotli, or even to design a novel SSSP algorithm for entropic edge-weights possibly inspired by the ideas in [15, 18].
- The block partitioning computation executed in Phase II (see Section 4) is based on the heuristic proposed in the compressor `Zopfli` that is iterative and heuristic, and has no mathematical guarantee. An interesting follow-up would be to deploy the result in [12] to compute a block partitioning in just one round, by optimizing the sum of the individual entropies of the generated blocks (hence, minimizing in some way the final output produced by the related Huffman codes). This has been already tried with some success in `Zopfli`, and thus it could be promising in Brotli because this step takes a considerable time of Phase II, i.e. about 9–38% of the total compression time as illustrated in Table 2.
- We have not dealt with the issues related to the 12 *quality levels* that Brotli offers to its users to trade compressed space for (de)compression time, and we did concentrate our study only on the last one, which offers the best compression ratio by involving all available optimizations. If we would plot the compression time/space trade-off curve achieved by various quality levels we would note that it is actually neither smooth nor predictable. One important issue would thus be to make it *smoother*, especially for the gap occurring between levels 9 and 10.
- The question of whether and how the Bicriteria scheme introduced in [9] can be applied to Brotli is worth investigating, provided that here we cannot use universal encoders and have to stick with the format of the compressed meta-blocks. How much can we relax the various techniques introduced in Phase II to increase Brotli's decompression speed without sacrificing much its compression ratio?

These ones and the other avenues of research inspired by the results shown in the previous pages are left to the creativity and problem solving ability of the readers.

REFERENCES

[1] Jyrki Alakuijala and Zoltan Szabadka. 2016. Brotli Compressed Data Format (RFC 7932). *IETF: Internet Engineering Task Force* (July 2016). https://tools.ietf.org/html/rfc7932

[2] Brian Allen and J. Ian Munro. 1978. Self-organizing binary search trees. *J. ACM* 25, 4 (1978), 526–535. https://doi.org/10.1145/322092.322094

[3] Michael A. Bender and Martin Farach-Colton. 2004. The Level Ancestor Problem simplified. *Theor. Comput. Sci.* 321, 1 (2004), 5–12.

[4] Michael Burrows and David J. Wheeler. 1994. *A block-sorting lossless data compression algorithm.* Technical Report. Digital Corp.

[5] Yann Collet. 2011. *The compressor LZ4.* https://en.wikipedia.org/wiki/LZ4_(compression_algorithm)

[6] Yann Collet. 2016. *The compressor ZStd.* http://www.zstd.net

[7] Lasse Collin. 2010. *XZ Utils.* https://tukaani.org/xz/

[8] Jarek Duda, Khalid Tahboub, Neeraj J. Gadgil, and Edward J. Delp. 2015. The use of asymmetric numeral systems as an accurate replacement for Huffman coding. In *2015 Picture Coding Symposium (PCS)*. 65–69.

[9] Andrea Farruggia, Paolo Ferragina, Antonio Frangioni, and Rossano Venturini. 2014. Bicriteria data compression. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014*. ACM-SIAM, 1582–1595.

[10] Andrea Farruggia, Paolo Ferragina, and Rossano Venturini. 2014. Bicriteria Data Compression: Efficient and Usable. In *European Symposium on Algorithms (ESA)*, Vol. 8737. Lecture Notes in Computer Science. Springer, 406–417.

[11] Paolo Ferragina, Raffaele Giancarlo, Giovanni Manzini, and Marinella Sciortino. 2005. Boosting textual compression in optimal linear time. *J. ACM* 52, 4 (2005), 688–713.
[12] Paolo Ferragina, Igor Nitto, and Rossano Venturini. 2011. On Optimally Partitioning a Text to Improve Its Compression. *Algorithmica* 61, 1 (2011), 51–74.
[13] Paolo Ferragina, Igor Nitto, and Rossano Venturini. 2013. On the bit-complexity of Lempel-Ziv compression. *SIAM J. Comput.* 42, 4 (2013), 1521–1541.
[14] Rich Geldreich. 2010. *The compressor LZHAM.* https://github.com/richgel999/lzham_codec
[15] Eran Halperin and Richard M. Karp. 2005. The minimum-entropy set cover problem. *Theor. Comput. Sci.* 348, 2-3 (2005), 240–250.
[16] Apple Inc. 2015. *The compressor LZFSE.* https://github.com/lzfse/lzfse
[17] Sanjay Jeff Dean, Steinar Ghemawat, and H. Gunderson. 2011. *The compressor Snappy.* https://en.wikipedia.org/wiki/Snappy_(compression)
[18] Shmuel T. Klein. 2000. Improving static compression schemes by alphabet extension. In *Proceedings of the 11th Combinatorial Pattern Matching Conference (CPM).* 210–221.
[19] Alistair Moffat. 1990. Implementing the PPM Data Compression Scheme. *IEEE Trans. on Communications* 38, 11 (1990), 1917–1921.
[20] Igor Pavlov. 1998. *The algorithm: Lempel-Ziv-Markov chain.* https://en.wikipedia.org/wiki/Lempel-Ziv-Markov_chain_algorithm
[21] David Salomon. 2007. *Data Compression: the Complete Reference, 4th Edition.* Springer Verlag.
[22] E. J. Schuegraf and H. S. Heaps. 1974. A comparison of algorithms for data base compression by use of fragments as language elements. *Information Storage and Retrieval* 10, 9-10 (1974), 309–319.
[23] Raimund Seidel and Cecilia R. Aragon. 1996. Randomized search trees. *Algorithmica* 16, 4/5 (1996), 464–497.
[24] Daniel Dominic Sleator and Robert Endre Tarjan. 1985. Self-Adjusting Binary Search Trees. *J. ACM* 32, 3 (1985), 652–686. https://doi.org/10.1145/3828.3835
[25] Andrew Turpin and Alistair Moffat. 2000. Housekeeping for prefix coding. *IEEE Trans. Communications* 48, 4 (2000), 622–628.
[26] Lode Vandevenne and Jyrki Alakuijala. 2013. *Zopfli.* https://github.com/google/zopfli
[27] I. H. Witten, A. Moffat, and T. C. Bell. 1999. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann Publishers. xxxi + 519 pages.
[28] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 23, 3 (1977), 337–343.
[29] Jacob Ziv and Abraham Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory* 24, 5 (1978), 530–536.

A ON THE COMPUTATIONAL DIFFICULTY OF PROBLEM MELZ

In order to keep the discussion simple, yet sufficiently significant, we restrict our attention to the case of an input string $\mathcal{S}$ whose alphabet Σ has size $\sigma = \Omega(|\mathcal{S}|)$ and consider a variant of the LZ77-parsing problem in which the first occurrence of every symbol c is encoded with the phrase $\langle 0, c\rangle$, whereas all other phrases of the parsing have the form $\langle d, \ell\rangle$, with $d > 0$. It is clear that the cost of encoding the former type of phrases is fixed and thus it does not enter in the optimization; moreover, it goes without saying that more technicalities are needed to deal with the case of a constant-sized alphabet and the presence of individual literals, but we defer this discussion to a follow-up theoretical paper. The main message of this section is to give the intuition that the more general MELZ is computationally difficult by showing that the simple, yet significant, variant above is indeed NP-hard.

The NP-hardness proof is based on a reduction from the *Minimum Entropy Set Cover* problem (MESC) introduced by Halperin and Karp [15]. An instance $\mathcal{I} = (\mathcal{U}, \mathcal{F})$ of MESC consists of a universe $\mathcal{U}$ of m elements $u_1, \ldots, u_m$ and a family $\mathcal{F}$ of k subsets $F_1, \ldots, F_k$ of $\mathcal{U}$. A *covering* f is a function $f : \mathcal{U} \to \mathcal{F}$ which *labels* each element in $\mathcal{U}$ with a subset in $\mathcal{F}$ that contains it.

DEFINITION 2. *Given a universe $\mathcal{U}$ and a family $\mathcal{F}$ of subsets of $\mathcal{U}$, the* Minimum-Entropy Set Cover problem *(MESC) asks for a cover f that minimizes the entropy H_0 of the sequence $(f(u_1), \ldots, f(u_m))$.*

Paraphrasing this definition, MESC asks for a labelling of the elements in $\mathcal{U}$ that can be encoded in minimum space via an entropic coder.

Having said this, our polynomial-time reduction takes an instance $\mathcal{I} = (\mathcal{U}, \mathcal{F})$ of MESC and instantiates:

- a string $\mathcal{S}$ of length $(2k+1)m$ over an alphabet Σ of size $\sigma = 3m$;
- a mapping from LZ77-parsings π of $\mathcal{S}$ to coverings f_π of $\mathcal{I}$. The mapping returns an optimal covering f_π for $\mathcal{I}$ if π is a solution for the MELZ problem instantiated over $\mathcal{S}$.

String $\mathcal{S}$ is defined as the concatenation of m widgets $W_1, \ldots, W_m$, one for each element u_i of $\mathcal{U}$, such that W_i is a string over the sub-alphabet $\Sigma_i = \{\sigma^i_Y, \sigma^i_N, \sigma^i_S\}$. All Σ_i are pairwise-disjoint, and the string-widget W_i is defined as follows:

$$W_i = \chi(i,k)\cdots\chi(i,1)\ \underbrace{\sigma^i_S \cdots \sigma^i_S}_{k \text{ times}}\ A(i)$$

where $A(i) = \sigma^i_Y$ and

$$\chi(i,j) = \begin{cases} \sigma^i_Y & u_i \in F_j \\ \sigma^i_N & \text{otherwise} \end{cases}$$

Let us call the symbols $\chi(i,j)$ the *characteristic symbols* for the sets F_j, the symbols σ^i_S the *separator symbols* and, finally, the last symbol $A(i)$ the *assignment symbol*. The following lemma states that $A(i)$ must be copied from within W_i and easily follows from the following observations: (i) each widget is expressed over a distinct alphabet, (ii) every element u_i belongs to at least one F_j (so it is $\chi(i,j) = \sigma^i_Y$), and (iii) recall that the LZ77-parsing π consists only of copy-phrases, because we have dropped from the analysis the first occurrences of all symbols given their fixed total cost $cost(\Sigma)$. Therefore, we have:

LEMMA 1. *For any LZ77-parsing π of $\mathcal{S}$ which uses only phrase copies, each assignment symbol $A(i)$ of $\mathcal{S}$ induces an LZ77-phrase in π which copies the assignment symbol $A(i)$ from some $\chi(i,K_i) = \sigma^i_Y$, in the same widget W_i, at a distance within the range $[k+1, 2k]$.*

Lemma 1 allows us to define our mapping as follows: $f_\pi(u_i) = F_{K_i}$. Moreover, given a covering f of $\mathcal{I}$ a LZ77-parsing π of the string $\mathcal{S}$ can be defined such that $f_\pi = f$: simply copy $A(i)$ from $\chi(i,j)$ whenever $f(u_i) = F_j$ and parse the rest of $\mathcal{S}$ using (say) the classic greedy strategy. In other words, our mapping from parsings to coverings is *surjective* in the set of coverings of $\mathcal{I}$. These simple observations are enough to prove the main result of this section:

THEOREM A.1. *Solving the variant of MELZ in which parsings are formed by phrase copies and first occurrences of every symbol (but not literal runs) is $\mathcal{NP}$-Hard whenever the string $\mathcal{S}$ is defined over an alphabet Σ with $|\Sigma| = \Theta(n)$.*

PROOF. We propose a reduction from MESC. Let Π be the set of all LZ77-parsings of $\mathcal{S}$ and let $\pi^\star$ be the (optimal) solution of the following minimization problem:

$$\min_{\pi\in\Pi} |\pi|\ (H_0(\mathsf{lens}(\pi)) + H_0(\mathsf{dists}(\pi))) \qquad (1)$$

We now show that $f_{\pi^\star}$ is an optimal covering of $\mathcal{I}$. Let us rewrite the term $H_0(\mathsf{dists}(\pi))$ as follows:

$$H_0(\mathsf{dists}(\pi)) = \sum_{d=1}^{k} \frac{\mathsf{occ}(d,\pi)}{|\pi|} \log \frac{|\pi|}{\mathsf{occ}(d,\pi)} + \sum_{d=k+1}^{2k} \frac{\mathsf{occ}(d,\pi)}{|\pi|} \log \frac{|\pi|}{\mathsf{occ}(d,\pi)}$$

The second summation can be rewritten as follows:

$$\begin{aligned}
\sum_{d=k+1}^{2k} \frac{\mathsf{occ}(d,\pi)}{|\pi|} \log \frac{|\pi|}{\mathsf{occ}(d,\pi)} &= \sum_{j=1}^{k} \frac{\mathsf{occ}(k+j,\pi)}{|\pi|} \log \frac{|\pi|}{\mathsf{occ}(k+j,\pi)} \\
&= \frac{m}{|\pi|} \left(\sum_{j=1}^{k} \frac{\mathsf{occ}(k+j,\pi)}{m} \left(\log \frac{m}{\mathsf{occ}(k+j,\pi)} \times \frac{|\pi|}{m} \right) \right) \\
&= \frac{m}{|\pi|} \left(\sum_{j=1}^{k} \frac{\mathsf{occ}(k+j,\pi)}{m} \left(\log \frac{m}{\mathsf{occ}(k+j,\pi)} + \log \frac{|\pi|}{m} \right) \right) \\
&= \frac{m}{|\pi|} \left(\log \frac{|\pi|}{m} + \sum_{j=1}^{k} \frac{\mathsf{occ}(k+j,\pi)}{m} \log \frac{m}{\mathsf{occ}(k+j,\pi)} \right) \\
&= \frac{m}{|\pi|} \left(\log \frac{|\pi|}{m} + H_0(f_\pi) \right)
\end{aligned}$$

where the last two equalities come from the observation that only the assignment symbols can be copied from distances in the range $[k+1,\, 2k]$ (Lemma 1), so that $\sum_{j=1}^{k} \mathsf{occ}(k+j,\pi) = m$. By substituting this into the minimization formula (1) above we get:

$$\min_{\pi \in \Pi} |\pi| \cdot \left(H_0(\mathsf{lens}(\pi)) + \sum_{d=1}^{k} \frac{\mathsf{occ}(d,\pi)}{|\pi|} \log \frac{|\pi|}{\mathsf{occ}(d,\pi)} \right) + m \log \frac{|\pi|}{m} + m\, H_0(f_\pi)$$

Observe that the term $m\, H_0(f(\pi))$ only depends on the distance components of the copies of the assignment symbols, which on the other hand do not affect the rest of the expression. So $\pi^\star$ is an LZ77-parsing that achieves the following bound:

$$\min_{\pi \in \Pi} \left(|\pi| \cdot \left(H_0(\mathsf{lens}(\pi)) + \sum_{d=1}^{k} \frac{\mathsf{occ}(d,\pi)}{|\pi|} \log \frac{|\pi|}{\mathsf{occ}(d,\pi)} \right) + m \log \frac{|\pi|}{m} \right) + \min_{\pi \in \Pi} H_0(f_\pi)$$

and, eventually, $\pi^\star$ must minimize (individually) the right expression $H_0(f_\pi)$ over all LZ77-parsings in Π. According to above, f is surjective in the set of coverings of $\mathcal{I}$, so that $f_{\pi^\star}$ is an optimal solution to MESC. □

Received *****; revised ****; final version ******; accepted ****