SIAM J. COMPUT.
Vol. 48, No. 5, pp. 1603–1642

BICRITERIA DATA COMPRESSION*

ANDREA FARRUGGIA†, PAOLO FERRAGINA†, ANTONIO FRANGIONI†, AND ROSSANO VENTURINI†

Abstract. Since the seminal work by Shannon, theoreticians have focused on designing compressors targeted at minimizing the output size without sacrificing much of the compression/decompression efficiency. On the other hand, software engineers have deployed several heuristics to implement compressors aimed at trading compressed space versus compression/decompression efficiency in order to match their application needs. In this paper we fill this gap by introducing the *bicriteria data-compression problem* that seeks to determine the shortest compressed file that can be decompressed in a given time bound. Then, inspired by modern data-storage applications, we instantiate the problem onto the family of Lempel–Ziv-based compressors (such as Snappy and LZ4) and solve it by combining in a novel and efficient way optimization techniques, string-matching data structures, and shortest path algorithms over properly (bi-)weighted graphs derived from the data-compression problem at hand. An extensive set of experiments complements our theoretical achievements by showing that the proposed algorithmic solution is very competitive with respect to state-of-the-art highly engineered compressors.

Key words. approximation algorithm, data compression, Lempel–Ziv 77, Pareto optimization

AMS subject classifications. 68P30, 90C27

DOI. 10.1137/17M1121457

1. Introduction.

The advent of massive datasets and the consequent design of high-performing distributed storage systems, such as Bigtable by Google [11], Cassandra by Facebook [7], and Hadoop by Apache, have reignited the interest of the scientific and engineering community towards the design of lossless data compressors that achieve effective compression ratio and very high decompression speed. The literature abounds with solutions for this problem, referred to as "compress once, decompress many times." These solutions can be cast into two main families: the compressors based on the Burrows–Wheeler transform (BWT) [10], and the ones based on the Lempel–Ziv parsing scheme [44, 45]. Algorithms are known in both families that require linear time in the input size, both for compressing and decompressing the data, and size of the compressed file that can be bound in terms of the kth order empirical entropy of the input [30, 44].

The compressors running behind real-world large-scale storage systems, however, are not derived from those scientific results. The reason is that theoretically efficient compressors are optimal in the RAM model, but they elicit many cache/IO misses on decompression. This poor behavior is most prominent in the BWT-based compressors, and it is also not negligible in the Lempel–Ziv (LZ)-based approaches. This motivated the software engineers to devise variants of Lempel and Ziv's original proposal (e.g., Snappy, Lz4) with the injection of several software tricks that have beneficial effects on memory-access locality. These compressors expanded further the known

*Received by the editors March 17, 2017; accepted for publication (in revised form) August 9, 2019; published electronically October 31, 2019. This paper combines and extends two conference papers that appeared in the proceedings of the ACM-SIAM Symposium on Discrete Algorithms (2014) and of the European Symposium on Algorithms (2014).
https://doi.org/10.1137/17M1121457
†Department of Computer Science, University of Pisa, 56123 Pisa, Italy (a.farruggia@di.unipi.it, paolo.ferragina@unipi.it, antonio.frangioni@unipi.it, rossano.venturini@unipi.it).

jungle of time/space trade-offs,[1] thus confronting software engineers with a difficult choice: either achieve effective/optimal compression ratios, at the possible sacrifice of decompression speed (as occurs in the known theory-based results [18, 19, 20]), or try to balance compression ratio *versus* decompression speed by adopting a plethora of programming tricks that actually waive any mathematical guarantees on their final performance (such as in Snappy, Lz4) or by adopting approaches that can only offer a rough asymptotic guarantee (such as in LZ-end, designed by Kreft and Navarro [31], which is more suited for highly compressible datasets).

In light of this dichotomy, it would be natural to ask for an algorithm that guarantees effective compression ratio and efficient decompression speed in hierarchical memories. In this paper, however, we aim for a more ambitious goal that is further motivated by the following two simple, yet challenging, questions:

1. Is it possible to obtain a slightly larger compressed file than the one achievable with BWT-based compressors while significantly improving on BWT's decompression time? This is a natural question arising in the context of distributed storage systems, such as the ones leading to the design of Snappy and Lz4.
2. Is it possible to obtain a compressed file that can be decompressed slightly slower than Snappy or Lz4 while significantly improving on their compressed space? This is a natural question in contexts where space occupancy is a major concern, e.g., tablets and mobile phones, and for which tools like Google's Brotli have been recently introduced.

By providing appropriate mathematical definitions for the two fuzzy expressions above—i.e., "slightly larger" and "slightly slower"—these two questions become pertinent and challenging in theory too. To this aim, we introduce the *bicriteria data compression* problem (BDCP): given an input file $\mathcal{S}$ and an upper bound $\mathcal{T}$ on its decompression time, determine a compressed version of $\mathcal{S}$ that minimizes the compressed space provided that it can be decompressed in $\mathcal{T}$ time. Symmetrically, we can exchange the role of time/space resources, and thus ask for the compressed version of $\mathcal{S}$ that minimizes the decompression time provided that the compressed space occupancy is within a fixed bound. Of course, in order to attack this problem in a principled way, we need to fix two ingredients: the class of compressed versions of $\mathcal{S}$ over which the bicriteria optimization will take place, and the computational model measuring the resources to be optimized.

For the former ingredient we select the class of LZ77-*based compressors*, which are dominant both in the theoretical setting (e.g., [12, 13, 17, 20, 26, 27]) and in the practical setting (e.g., Gzip, 7zip, Snappy, Lz4 [29, 31, 43]). In section 2 we show that BDCP formulated over LZ77-based compressors is a theoretically sound problem because there exists an *infinite class of strings* that can be parsed in many different ways and can offer a wide spectrum of time/space trade-offs in which small variations in the usage of one resource (e.g., time) may induce arbitrarily large variations in the usage of the other resource (e.g., space).

For the latter ingredient we take inspiration from several known models of computation that abstract multilevel memory hierarchies and the fetching of contiguous memory words (Aggarwal and colleagues [1, 2], Alpern et al. [5], Luccio and Pagli [35], and Vitter and Shriver [42]). In these models the cost of fetching a word at address x takes $f(x)$ time, where $f(x)$ is a nondecreasing, polynomially bounded function (e.g., $f(x) = \lceil \log x \rceil$ and $f(x) = x^{\Theta(1)}$). Some of these models offer also a *block-copy* operation, in which a sequence of ℓ consecutive words can be copied from memory

[1] See, e.g., http://mattmahoney.net/dc/.

location x to memory location y (with $x \geq y$) in time $f(x) + \ell$. We remark that, in our scenario, this model is more proper than the frequently adopted two-level memory model [4], because we care to differentiate between contiguous and random accesses to memory/disk blocks, this being a feature heavily exploited in the implementation of modern compressors [14].

Given these two ingredients, we devise a formal framework that allows us to analyze any LZ77-parsing scheme in terms of both the space occupancy (in bits) of the compressed file, and the time cost of its decompression, taking into account the underlying memory hierarchy. More specifically, we start from the model proposed by Ferragina, Nitto, and Venturini [20] that represents the input text $\mathcal{S}$ via a special weighted directed acyclic graph (DAG) consisting of $n = |\mathcal{S}|$ nodes, one per character of $\mathcal{S}$, and $m = \mathcal{O}(n^2)$ edges, one per possible phrase in the LZ77-parsing of $\mathcal{S}$. We augment this graph by labeling each edge with two costs: a *time cost*, which accounts for the time to decompress an LZ77-phrase (derived according to the hierarchical-memory model mentioned above), and a *space cost*, which accounts for the number of bits needed to store the LZ77-phrase associated to that edge (derived according to the integer encoder adopted in the compressor). Every path π from node 1 to node n in $\mathcal{G}$ (hereafter, "$1n$-path") corresponds to an LZ77-parsing of the input file $\mathcal{S}$, whose *compressed-space occupancy* is given by the sum of the *space costs* of π's edges (say, $s(\pi)$) and whose *decompression time* is given by the sum of the *time costs* of π's edges (say, $t(\pi)$). As a result, we are able to reformulate BDCP as a *weight-constrained shortest path* problem (WCSPP) over the weighted DAG $\mathcal{G}$, seeking for the $1n$-path π that minimizes the compressed-space occupancy $s(\pi)$ among all those $1n$-paths whose decompression time is $t(\pi) \leq \mathcal{T}$.

Due to its vast range of applications WCSPP received a great deal of attention from the optimization community; see, e.g., the work by Mehlhorn and Ziegelmann [38] and references therein. It is an $\mathcal{NP}$-hard problem, even when the DAG $\mathcal{G}$ has positive costs [15, 21], but not strongly $\mathcal{NP}$-hard as it can be solved in pseudopolynomial $\mathcal{O}(m\mathcal{T})$ time via dynamic programming [33]. Our version of the WCSPP problem has the parameters m and $\mathcal{T}$ bounded by $\mathcal{O}(n \log n)$ (see section 2), so it can be solved in polynomial time $\mathcal{O}(m\mathcal{T}) = \mathcal{O}(n^2 \log^2 n)$ and $\mathcal{O}(n^2 \log n)$ space. Unfortunately these bounds are unacceptable in practice, because $n^2 \approx 2^{64}$ just for one gigabyte of data to be compressed.

The first algorithmic contribution of this paper is to exploit the structural properties of the weighted DAG in order to design an algorithm that approximately solves the corresponding WCSPP in $\mathcal{O}(n \log^2 n)$ time and $\mathcal{O}(n)$ working space. The approximation is *additive*, that is, our algorithm determines an LZ77-parsing of $\mathcal{S}$ whose decompression time is $\leq \mathcal{T} + 2\, t_{\max}$ and whose compressed space is just at most $s_{\max}$ bits more than the optimal one, where $t_{\max}$ and $s_{\max}$ are, respectively, the maximum time cost and the maximum space cost of any edge in the DAG. Notably, the values of $s_{\max}$ and $t_{\max}$ are logarithmic in n so those additive terms are negligible (see section 2). We remark here that this type of additive approximation is clearly related to the *bicriteria-approximation* introduced by Marathe et al. [36], and it is more desirable than the "classic" (α, β)-approximation [34], which is multiplicative. A further peculiarity of our solution is that the additive approximation is used to speed up the solution from $\Omega(n^2)$ time, which is poly-time but unusable in practice, to the more practical $\mathcal{O}(n \log^2 n)$ time. In section 4.4 we show that our solution can be easily generalized over graphs satisfying some simple properties, which might be of independent interest.

The bicriteria strategy proposed in this paper deploys as a subroutine the bit-

optimal LZ77-compressor devised by Ferragina, Nitto, and Venturini [20]. Unfortunately that algorithm, albeit asymptotically optimal in the RAM model, is not efficient in practice due to its expensive pattern of memory accesses. As a second algorithmic contribution, we propose a novel asymptotically optimal LZ77-compressor, hereafter named LzOpt, that is significantly faster in practice because it exploits simpler data structures and accesses memory in a sequential fashion (see section 3.2). This represents an important step in closing the compression-time gap between LzOpt and the widely used compressors such as Gzip, Bzip2, and Lzma2.

As a third, and last, contribution of this paper we experimentally evaluate both LzOpt and the resulting bicriteria compressor, hereafter named Bc-Zip, under various experimental scenarios (see section 5). We investigate many algorithmic and implementation issues on several datasets of different types. We also compare their performance with an ample set of both classic (LZ-based, PPM-based, and BWT-based) and engineered (Snappy, Lz4, Lzma2, and Ppmd) compressors. The whole experimental setting originated by this effort, consisting of the datasets and the C++ code of Bc-Zip, has been made available to the scientific community to allow the replication of those experiments and the possible testing of new solutions (see https://github.com/farruggia/bc-zip). Eventually, the experiments reveal two key aspects:

1. The two questions posed at the beginning of the paper are indeed pertinent, because real texts can be parsed in many different ways, offering a wide spectrum of time/space trade-offs in which small variations in the usage of one resource (e.g., time) may induce arbitrary large variations in the usage of the other resource (e.g., space). This motivates the introduction of our novel parsing strategy.
2. Our parsing strategy actually dominates all the highly engineered competitors, by exhibiting decompression speeds close to or better than those achieved by Snappy and Lz4 (i.e., the fastest ones) and compression ratios close to those achieved by BWT-based and Lzma2 compressors (i.e., the more succinct ones).

The paper is organized as follows. In section 2 we briefly introduce the LZ77 parsing scheme, we define the models for estimating the compressed space (section 2.1) and decompression time (section 2.2) of a given parsing, and we show an infinite family of strings that exhibit many different time/space trade-offs (section 2.3). In section 3 we illustrate the bit-optimal parsing introduced by Ferragina, Nitto, and Venturini [20]. In particular, in section 3.1 we briefly review their work, while in section 3.2 we illustrate our algorithmic improvements aimed at making the approach more efficient and practical. In section 4 we illustrate our novel bicriteria strategy and obtain the approximation result. Section 5 details our experiments and some interesting implementation choices: section 5.1 illustrates how to extend the bit-optimal and bicriteria strategies to take into account *literals strings*, i.e., LZ77-phrases that represent an uncompressed sequence of characters, while section 5.2 illustrates the performance of the new bit-optimal parser illustrated in section 3.2. Section 5.3 integrates section 2.2 with additional details about deriving a time model in an automated fashion on an actual machine, as well as showing its accuracy on our experimental setting. Section 5.4 illustrates the performance of the bicriteria parser on our dataset. Finally, in section 6 we sum up this work and point out some interesting future research on the subject.

2. On the LZ77-parsing. Let $\mathcal{S}$ be a string of length n built over an alphabet $\Sigma = [\sigma]$ and terminated by a special character. We denote by $\mathcal{S}[i]$ the ith character

of $\mathcal{S}$ and by $\mathcal{S}[i,j]$ the substring ranging from position i to position j (included). The compression algorithm LZ77 works by *parsing* the input string $\mathcal{S}$ into phrases $p_1, \ldots, p_k$ such that phrase p_i can be either a single character or a substring that occurs also in the prefix $p_1 \cdots p_{i-1}$, and thus can be copied from there. Once the 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. Every first occurrence of a new character c is encoded as $\langle 0, c \rangle$. These pairs are compressed into *codewords* via variable-length integer encoders that eventually produce the compressed output of $\mathcal{S}$ as a sequence of bits. Among all possible parsing strategies, the *greedy parsing* is widely adopted: it chooses p_i as the longest prefix of the remaining suffix of $\mathcal{S}$. This is optimal whenever the goal is to minimize the number of generated phrases or, equivalently, the size of the compressed file under the assumption that these phrases are encoded with the same number of bits; however, if phrases are encoded with a variable number of bits, then the greedy approach may be highly suboptimal [20], and this justifies the study and results introduced in this paper.

2.1. Modeling space occupancy. An LZ77-phrase $\langle d, \ell \rangle$ is typically compressed by using two distinct (universal) integer encoders $\mathsf{enc}_{\text{dist}}$ and $\mathsf{enc}_{\text{len}}$, since distances d and lengths ℓ have different distributions in $\mathcal{S}$. We use $s(d, \ell)$ to denote the length in bits of the encoding of $\langle d, \ell \rangle$, that is, $s(d, \ell) = |\mathsf{enc}_{\text{dist}}(d)| + |\mathsf{enc}_{\text{len}}(\ell)|$. We assume that a phrase consisting of one single character is represented by a fixed number of bits. We restrict our attention to variable-length integer encoders that emit longer codewords for bigger integers. This property is called the *nondecreasing cost property* and is stated as follows.

PROPERTY 2.1. *An integer encoder* enc *satisfies the nondecreasing cost property if* $|\mathsf{enc}(n)| \leq |\mathsf{enc}(n')|$ *for all positive integers* $n \leq n'$.

We also assume that these encoders must be *stateless*, that is, they must always encode the same integer with the same bit sequence. These two assumptions are not restrictive because they encompass all known universal encoders, such as truncated binary, Elias' Gamma and Delta [16, 22], and Lz4's encoder.[2]

In the following (see Table 2.1 for a recap of the main notation), we denote by $s(\mathcal{P}) = \sum_{\langle d,\ell \rangle \in \mathcal{P}} s(d, \ell)$ the length in bits of the compressed output generated according to the LZ77-parsing $\mathcal{P}$. We also denote by s_{costs} the number of distinct values assumed by $s(d, \ell)$ when $d, \ell \leq n$; this value is relevant as it affects the time complexity of the parsing strategies presented in this paper. Further, we denote as Q_{dist} the number of times the value $|\mathsf{enc}_{\text{dist}}(d)|$ changes when $d = 1, \ldots, n$; value Q_{len} is defined in a similar way. Under the assumption that both $\mathsf{enc}_{\text{dist}}$ and $\mathsf{enc}_{\text{len}}$ satisfy Property 2.1, it holds that $s_{\text{costs}} = Q_{\text{dist}} + Q_{\text{len}}$. Most interesting integer encoders, such as those listed above, encode integers in a number of bits proportional to their logarithm, and, thus, both Q_{dist} and Q_{len} are $\mathcal{O}(\log n)$.

2.2. Modeling decompression time. The aim of this section is to define a model that can accurately predict the decompression time of an LZ77-compressed text in modern machines, characterized by memory hierarchies equipped with cache-

[2]We notice that other well-known integer encoders that do not satisfy the nondecreasing cost property, such as PForDelta and Simple9, are indeed not universal and, crucially, are designed to be effective only in some specific situations, such as the encoding of inverted lists. Since these settings have very specific assumptions about symbol distribution that are generally *not* satisfied by LZ77-parsings, they are not examined in this paper.

TABLE 2.1
Summary of main notation.

Name	Definition
$\mathcal{S}$	A (null-terminated) document to be compressed.
n	Length of $\mathcal{S}$ (end-of-text character included).
$\mathcal{S}[i]$	The ith character of $\mathcal{S}$.
$\mathcal{S}[i,j]$	Substring of $\mathcal{S}$ starting from $\mathcal{S}[i]$ until $\mathcal{S}[j]$ (included).
$\langle d,\ell\rangle$	An LZ77-phrase of length ℓ that can be copied at distance d.
$\langle 0,c\rangle$	An LZ77-phrase that represents the single character c.
$\langle \ell,\alpha\rangle_L$	An LZ77-phrase that represents a substring α of length ℓ.
$t(d)$	Amount of time spent in accessing the first character of a copy at distance d.
$s(d,\ell)$	The length in bits of the encoding of $\langle d,\ell\rangle$.
$t(d,\ell)$	The time needed to decompress the LZ77-phrase $\langle d,\ell\rangle$.
$s(\pi)$	The space occupancy of parsing π.
$t(\pi)$	The time needed to decompress the parsing π.
$s_{\max}$	The maximum space occupancy (in bits) of any LZ77-phrase of $\mathcal{S}$.
$t_{\max}$	The maximum time taken to decompress a LZ77-phrase of $\mathcal{S}$.
s_{costs}	The number of distinct values that may be assumed by $s(d,\ell)$ when $d \le n$, $\ell \le n$.
t_{costs}	The number of distinct values that may be assumed by $t(d,\ell)$ when $d \le n$, $\ell \le n$.
Q	Number of cost classes for the integer encoder enc (may depend on n).
$L(k)$	Codeword length (in bits) of integers belonging to the kth cost class of enc.
$M(k)$	Largest integer encodable by the kth cost class of enc.
$E(k)$	Number of integers encodable within the kth cost class (i.e., $E(k) = M(k) - M(k-1)$).
$W(k,j)$	Defined as the jth block of $E(k)$ contiguous nodes in the graph $\mathcal{G}$.
$\mathsf{Sa}[1,n]$	The suffix array of $\mathcal{S}$; hence $\mathsf{Sa}[i]$ is the ith lexicographically smallest suffix of $\mathcal{S}$.
$\mathsf{Isa}[1,n]$	The inverse suffix array, so $\mathsf{Isa}[j]$ is the (lexicographic) position of $\mathcal{S}[j,n]$ among all text suffixes, hence in Sa (i.e., $j = \mathsf{Sa}[\mathsf{Isa}[j]]$).

prefetching strategies and sophisticated control-flow predictive processors. In a fashion analogous to the previous section, we assume that decoding a character $\langle 0,c\rangle$ takes constant time $t_{\mathcal{C}}$, while function $t(d,\ell)$ models the time spent in decoding an LZ77-phrase $\langle d,\ell\rangle$ and performing the phrase copy. The time needed to decompress parsing $\mathcal{P}$ (and so reconstruct $\mathcal{S}$) is thus estimated as $t(\mathcal{P}) = \sum_{\langle d,\ell\rangle\in\mathcal{P}} t(d,\ell)$.

Modeling the time spent in *decoding* an LZ77-phrase $\langle d,\ell\rangle$, denoted as $t_{\mathcal{D}}(d,\ell)$, is easy, as it is either constant or proportional to $s(d,\ell)$. On the other hand, modeling the time spent in *copying* is trickier, as it needs to take into account memory hierarchies. To do this, we take inspiration from several known hierarchical-memory models (Aggarwal and colleagues [1, 2], Alpern et al. [5], Luccio and Pagli [35], and Vitter and Shriver [42]). Let $t(d)$ be the access latency of the fastest (smallest) memory level that can hold at least d characters: we assume that accessing a character at distance d takes time $t(d)$. However, the time spent in copying ℓ characters going back d positions in $\mathcal{S}$ is highly influenced by *prefetching* and the way memory is laid out. Indeed, memory hierarchies in modern machines consist of multiple levels of memory ("caches"), where each cache-level C_i logically partitions the memory address space into cache-lines of size L_i. When an address x is requested to a cache of level i, the whole cache-line containing address x is transferred; hence, reading ℓ characters from cache C_i needs only $\lceil \ell/L_i \rceil$ accesses to cache C_i. Moreover, *cache-prefetching* is typically used to lower even further the time needed to read chunks of memory. In fact, when a sequence of cache misses to contiguous cache lines is triggered by the

processor, it instructs the cache to "stream" adjacent lines without an explicit processor request. After prefetching takes place, accessing subsequent characters from the cache C_i has the same cost as reading them from the closest (fastest) cache. The number of cache misses needed to trigger prefetching is hardware specific and difficult to estimate in advance, but it is usually two [14]. The cost of reading ℓ *contiguous* characters from a given memory level C_i is thus the cost of reading ℓ characters from the fastest memory level plus *at most* two cache misses in C_i, which we estimate with the term $n_M(\ell) \leq 2$.

Summing up, processing a phrase $\langle d, \ell \rangle$ takes time

$$t(d,\ell) = \begin{cases} t_{\mathcal{D}}(d,\ell) + n_M(\ell)\, t(d) + \ell\, t_{\mathcal{C}} & \text{if } \ell > 0, \\ t_{\mathcal{C}} & \text{if } \ell = 0. \end{cases}$$

Let us now estimate t_{costs}, the number of times $t(d,\ell)$ changes when $d, \ell \leq n$. Analogously to s_{costs}, this term affects the time complexity of the parsing strategies proposed in this paper, and therefore it is desirable that it is as low as possible.

Since $t(d,\ell)$ has an additive term $\ell\, t_{\mathcal{C}}$, t_{costs} is $\mathcal{O}(n)$. However, the overall contribution of this term to $t(\mathcal{P})$ is always $n\, t_{\mathcal{C}}$, so it can be taken into account by removing it from the definition of $t(d,\ell)$ and by setting the time bound as $\mathcal{T} - n\, t_{\mathcal{C}}$. As previously remarked, the term $t_{\mathcal{D}}(d,\ell)$ is proportional to the codeword length $s(d,\ell)$; hence, by definition, it changes s_{costs} times when $d, \ell \leq n$. Furthermore it is $n_M(\ell) \leq 2$ and $t(d) = \mathcal{O}(\log n)$ because sizes in hierarchical memories grow exponentially. Thus it follows that $t_{\text{costs}} = \mathcal{O}(s_{\text{costs}} + \log n)$.

Further details on the actual evaluation of the various parameters, as well as its actual decompression-time accuracy, are given in section 5.3.

2.3. Pathological strings: Trade-offs in space and time. We are interested in a parsing strategy that optimizes two different criteria, namely *decompression time* and *compressed space*. Unfortunately, usually there is no parsing that optimizes *both* of them, but, instead, there are several parsings that offer distinct time/space *trade-offs*. We thus have to turn our attention to *Pareto-optimal* parsings, i.e., parsings that are not *dominated* by any other parsing by being strictly worse on one account and not better on the other. We now show that there exists an infinite family of strings whose Pareto-optimal parsings exhibit significant differences in their decompression time versus compressed space.

Let us assume, for simplicity's sake, that each codeword takes constant space; moreover, let us also assume that memory is arranged in just two levels: a fastest level (of size c) with negligible access time and a slowest level (of unbounded size) with substantial access time. We construct our pathological input string $\mathcal{S}$ as follows. Let $\Sigma = \Sigma' \cup \$$, with \$ a character not in Σ', and let P be a string of length at most c, drawn over Σ', whose greedy LZ77-parsing takes k phrases. For any $i \geq 0$, let B_i be the string $\$^{c+i}P$. Set $\mathcal{S} = B_0 B_1 \dots B_m$. Since the length of run of \$s increases as i increases, and since character \$ does not occur in P, no pair of consecutive strings B_i and B_{i+1} can be part of the same LZ77-phrase. Moreover, we have two alternatives in parsing each B_i, with $i \geq 1$: either (i) we copy everything inside B_i, a strategy that takes $2 + k$ phrases of distance at most c and no cache miss at decompression time, or (ii) we copy from the previous string B_{i-1}, taking two phrases (a copy and a single character \$) and one cache miss at decompression time. There are m Pareto-optimal parsings of $\mathcal{S}$ obtained by choosing either the first or the second of the two alternatives just outlined for each string $B_{i \geq 1}$: given a positive number $\widetilde{m} \leq m$ of cache misses, the parsing that only copies blocks $B_{m-\widetilde{m}+1}, \dots, B_m$ has a smaller compressed space

than any other parsing with the same number $\widetilde{m}$ of cache misses. At one extreme, the parser always chooses alternative (i), thus obtaining a parsing with $m(2+k)$ phrases that is decompressible with no cache misses. At the other extreme, the parser always prefers alternative (ii), thus obtaining a parsing with $2+k+2m$ phrases that is decompressible with m cache misses (notice that block B_0 cannot be copied). A whole range of Pareto-optimal parsings stands between these two extremes, allowing us to trade decompression speed for space occupancy. In particular, one can save k phrases at the cost of one more cache miss, where k is a value that can be varied by choosing different strings P. The bicriteria strategy, illustrated in section 4, allows us to efficiently choose any of these trade-offs.

3. On the bit-optimal compression. In this section we discuss the *bit-optimal* LZ77*-parsing* problem (BLPP), defined as follows: given a text $\mathcal{S}$ and two integer encoders $\mathsf{enc}_{\text{dist}}$ and $\mathsf{enc}_{\text{len}}$, determine the minimal-space LZ77-parsing of $\mathcal{S}$ provided that all phrases are encoded with $\mathsf{enc}_{\text{dist}}$ (for the distance component) and $\mathsf{enc}_{\text{len}}$ (for the length component). Ferragina, Nitto, and Venturini [20] have shown an (asymptotically) efficient algorithm for this problem, proving the following theorem.

THEOREM 3.1. *Given a string $\mathcal{S}$ and two integer-encoding functions enc_{dist} and enc_{len} that satisfy Property* 2.1*, there exists an algorithm to compute the* bit-optimal LZ77-parsing *of $\mathcal{S}$ in $\mathcal{O}(n \cdot s_{costs})$ time and $\mathcal{O}(n)$ words of space.*

In this section we illustrate a novel BLPP algorithm that is much simpler, and therefore more efficient in practice, while still achieving the same time/space asymptotic complexities stated in the theorem above. Since BLPP is a key step for solving the bicriteria data compression problem (BDCP), these improvements are crucial to obtain an efficient bicriteria parser as well.

Our new algorithm relies on the same optimization framework devised by Ferragina, Nitto, and Venturini [20]. For the sake of clarity, we illustrate that framework in section 3.1, restating some lemmas and theorems proved in that paper. Briefly, the authors show how to reduce the BLPP to a *shortest path problem* on a DAG $\mathcal{G}$ with one node per character and one edge for every LZ77-phrase. However, even though the shortest path problem on DAGs can be solved linearly in the size of the graph, a straight resolution of BLPP would be inefficient because $\mathcal{G}$ might consist of $\mathcal{O}(n^2)$ edges. The authors solve this problem in two steps: first, $\mathcal{G}$ is pruned to a graph $\widetilde{\mathcal{G}}$ with just $\mathcal{O}(n \cdot s_{\text{costs}})$ edges, still retaining at least one optimal solution; second, the graph $\widetilde{\mathcal{G}}$ is generated *on-the-fly* in constant amortized time per edge, so that the shortest path can be computed in the time and space bounds stated in Theorem 3.1.

We notice that the codewords generated by the most interesting universal integer encoders have logarithmic length (i.e., $s_{\text{costs}} = \mathcal{O}(\log n)$), so the algorithm of Theorem 3.1 solves BLPP in $\mathcal{O}(n \log n)$ time. However, this algorithm requires the construction of suffix arrays and compact tries of several substrings of $\mathcal{S}$ (possibly transformed in proper ways), which are then accessed via memory patterns that make the algorithm pretty inefficient in practice.

In section 3.2 we show our new algorithm for the second step above, which is where our novel solution differs from the known one [20]. Our algorithm only performs scans over a few lists, so it is much simpler and therefore much more efficient in practice while simultaneously achieving the same time/space asymptotic complexities stated in Theorem 3.1.

3.1. Known results: A graph-based approach. Given an input string $\mathcal{S}$ of length n, BLPP is reduced to a shortest path problem over a weighted DAG $\mathcal{G}$ that

consists of $n+1$ nodes, one per input character plus a sink node, and m edges, one per possible LZ77-phrase in $\mathcal{S}$. In particular there are two different types of edges to consider:

(i) edge $(i, i+1)$, which represents the case of the single-character phrase $\mathcal{S}[i]$, and
(ii) edge (i, j), with $j = i + \ell > i + 1$, which represents the substring $\mathcal{S}[i, i + \ell - 1]$ that also occurs previously in $\mathcal{S}$.

This construction is similar to the one originally proposed by Schuegraf and Heaps [41]. By definition, every path π in this graph is mapped to a distinct parsing $\mathcal{P}$, and vice versa. An edge (i', j') is said to be *nested* into an edge (i, j) if $i \leq i' < j' \leq j$. A peculiar property of $\mathcal{G}$ that is extensively exploited throughout this paper is that its set of edges is closed under nesting.

PROPERTY 3.2. *Given an edge (i, j) of $\mathcal{G}$, any edge nested in (i, j) is an edge of $\mathcal{G}$.*

This property follows quite easily from the prefix/suffix-complete property of the LZ77 dictionary and from the bijection between LZ77-phrases and edges of $\mathcal{G}$. Since each edge (i, j) of $\mathcal{G}$ is associated to an LZ77-phrase, it can be weighted with the length, in bits, of its codeword. In particular,

(i) any edge $(i, i+1)$, a single character $\langle 0, \mathcal{S}[i] \rangle$, can be assumed to have constant cost;
(ii) for any other edge (i, j) with $j > i + 1$, a copy of length $\ell = j - i$ copying from some position $i - d$, is instead weighted with the value $s_{\mathcal{G}}(i, j) = s(d, \ell) = |\mathsf{enc}(d)| + |\mathsf{enc}(\ell)|$, where the notation $\langle d, \ell \rangle$ is used to denote the associated codeword.

As it currently stands, the definition of $\mathcal{G}$ is ambiguous whenever there exists a substring $\mathcal{S}[i, j-1]$ occurring more than once previously in the text: in this case, phrase $\mathcal{S}[i, j-1]$ references more than one string in the LZ77 dictionary. We fix this ambiguity by selecting one phrase $\langle d, \ell \rangle$ among those that minimize the expression $s(d, \ell)$. Hence, the space cost $s_{\mathcal{G}}(\pi)$ of any $1n$-path π (from node 1 to $n+1$), defined as the sum of the costs of its edges, is the compressed size $s(\mathcal{P})$ in bits of the associated parsing $\mathcal{P}$. Computing the bit-optimal parsing of $\mathcal{S}$ is thus equivalent to finding a shortest path on the graph $\mathcal{G}$. In the following we drop the subscript from $s_{\mathcal{G}}$ whenever this does not introduce ambiguities. Since $\mathcal{G}$ is a DAG, computing the shortest $1n$-path takes $\mathcal{O}(m+n)$ time and space; unfortunately, there are strings for which $m = \Theta(n^2)$ (e.g., $\mathcal{S} = a^n$ generates one edge per substring of $\mathcal{S}$). This means that a naive implementation of the shortest path algorithm would not be practical even for files of a few tens of MiBs. Two ideas have been deployed to solve this problem [20]:

1. Reduce $\mathcal{G}$ to an asymptotically smaller subset with only $\mathcal{O}(n \cdot s_{\text{costs}})$ edges in which the bit-optimal path is preserved.
2. Generate on-the-fly the edges in the pruned graph by means of an algorithm, called forward star generation (FSG), that takes $\mathcal{O}(1)$ amortized time per edge and optimal $\mathcal{O}(n)$ words of space. This is where our novel solution, illustrated in section 3.2, differs.

Pruning the graph. We now illustrate how to construct a reduced subgraph $\widetilde{\mathcal{G}}$ with only $\mathcal{O}(n \cdot s_{\text{costs}})$ edges that retains at least one optimal $1n$-path of $\mathcal{G}$. It is easy to show that if both $\mathsf{enc}_{\text{dist}}$ and $\mathsf{enc}_{\text{len}}$ satisfy the nondecreasing property, then all edges outgoing from a node have nondecreasing space costs when listed by increasing endpoint, from left to right.

LEMMA 3.3. *If both* enc_{dist} *and* enc_{len} *satisfy Property* 2.1*, then, for any node* i *and for any couple of nodes* $j < k$ *such that* $(i,j), (i,k) \in \mathcal{G}$*, it holds that* $s(i,j) \leq s(i,k)$.

This monotonicity property for the space costs of $\mathcal{G}$'s edges leads to the notion of *maximality* of an edge.

DEFINITION 3.4. *An edge* $e = (i,j)$ *is said to be* maximal *if and only if either the (next) edge* $e' = (i, j+1)$ *does not exist, or it does exist but* $s(i,j) < s(i,j+1)$.

Recall that s_{costs} is the number of distinct values assumed by $s(d,\ell)$ for $d, \ell \leq n$. By definition, every node has at most s_{costs} maximal outgoing edges. The following lemma shows that, for any path π, its "nested" paths do exist and have lower space cost.

LEMMA 3.5. *For each triple of nodes* $i < i' < j$*, and for each path* π *from* i *to* j*, there exists a path* π' *from* i' *to* j *such that* $s(\pi') \leq s(\pi)$.

The lemma, used with $j = n$, allows us to "push" nonmaximal edges to the right by iteratively replacing nonmaximal edges with maximal ones without augmenting the time and space costs of the path. By exploiting this fact, Theorem 3.6 shows that the search of shortest paths in $\mathcal{G}$ can be limited to paths composed of maximal edges only.

THEOREM 3.6. *For any* $1n$*-path* π *there exists a* $1n$*-path* $\pi^\star$ *composed of maximal edges only and such that the space cost of* $\pi^\star$ *is not worse than the space cost of* π*, i.e.,* $s(\pi^\star) \leq s(\pi)$.

Proof. We show that any $1n$-path π containing nonmaximal edges can be turned into a $1n$-path π' containing maximal edges only. Take the leftmost nonmaximal edge in π, say (v,w), and denote by π_v (resp., π_w) the prefix (resp., suffix) of path π ending in v (resp., starting from w). By definition of maximality, there must exist a maximal edge (v,z), with $z > w$, such that $s(v,z) = s(v,w)$. We can apply Lemma 3.5 to the triple (w,z,n) and thus derive a path μ from z to n such that $s(\mu) \leq s(\pi_w)$. We then construct the $1n$-path π'' by connecting the subpath π_v, the maximal edge (v,z), and the path μ: using Lemma 3.5 one readily shows that the space cost of π'' is not larger than that of π. The key result is that we pushed right the leftmost nonmaximal edge (if any) that must now occur (if ever) within μ; by iterating this argument we get the thesis. ∎

Hence, the pruned graph $\widetilde{\mathcal{G}}$ obtained by keeping only maximal edges in $\mathcal{G}$ has $\mathcal{O}(n \cdot s_{\text{costs}})$ edges. Most interesting integer encoders, such as truncated binary, Elias' Gamma and Delta [16], Golomb [22], and Lz4's encoder, encode integers in a number of bits proportional to their logarithm; thus $s_{\text{costs}} = \mathcal{O}(\log n)$ in practical settings, so $\widetilde{\mathcal{G}}$ is asymptotically sparser than $\mathcal{G}$, having only $\mathcal{O}(n \log n)$ edges.

THEOREM 3.7. *The subgraph* $\widetilde{\mathcal{G}}$*, defined by retaining only the maximal edges of* $\mathcal{G}$*, preserves at least one shortest* $1n$*-path of* $\mathcal{G}$ *and has* $\mathcal{O}(n \log n)$ *edges when using encoders with a logarithmically growing number of cost classes.*

The on-the-fly generation idea. As discussed previously, the pruning strategy consists of retaining only the maximal edges of $\mathcal{G}$, that is, every edge of maximum length among those of equal cost. Now we describe the basic ideas underlying the generation of these maximal edges incrementally along with the shortest path computation in $\mathcal{O}(1)$ amortized time per edge and only $\mathcal{O}(n)$ auxiliary space. We call this task the *forward star generation* (FSG). The efficient generation of the maxi-

mal edges needs to characterize them in a slightly different way. Let us assume, for simplicity's sake, that enc is used to encode both distances and lengths (that is, $\mathsf{enc} = \mathsf{enc}_{\text{dist}} = \mathsf{enc}_{\text{len}}$). If the function $|\mathsf{enc}(x)|$ changes only Q times when $x \leq n$, then enc induces a partitioning of the range $[1, n]$ (namely, the range of potential copy-distances and copy-lengths of LZ77-phrases in $\mathcal{S}$) into a sequence of Q subranges, such that all the integers in the kth subrange are represented by enc with codewords of length $L(k)$ bits, with $L(k) < L(k+1)$. We refer to such subranges as *cost classes* of enc because the integers in the same cost class are encoded by enc with the same number of bits (hence, they have the same cost in bits). If we denote by $M(k)$ the largest integer in the kth cost class and set $M(0) = 0$, then the subrange can be written as $[M(k-1)+1, M(k)]$. This implies that its size is $E(k) = M(k) - M(k-1)$, which indeed equals the number of integers represented with *exactly* $L(k)$ bits. Overall this implies that the compression of a phrase is unaffected whenever we change its copy-distance or copy-length with integers falling in the same cost class.

Moreover, notice that an edge (i, j) associated with a phrase $\langle d, \ell \rangle$ is *maximal* if it is the longest edge taking *exactly* $s(d, \ell) = |\mathsf{enc}(d)| + |\mathsf{enc}(\ell)|$ bits. A maximal edge can be either *d-maximal* or *ℓ-maximal* (or both): it is *d-maximal* if it is the longest edge taking exactly $|\mathsf{enc}(d)|$ bits for representing its distance component; ℓ-maximality is defined similarly.

Computing the d-maximal edges is the hardest task because every ℓ-maximal edge that is not also a d-maximal edge can be obtained in $\mathcal{O}(1)$ time by "shortening" a longer d-maximal edge. Let us illustrate this with an example: let $p = (i, j)$ and $q = (i, k)$ be two consecutive d-maximal edges outgoing from vertex i with $j < k$, and let $\langle d_p, \ell_p \rangle$ and $\langle d_q, \ell_q \rangle$ be their associated LZ77-phrases; then, all ℓ-maximal edges outgoing from vertex i whose endpoints fall between j and k have an associated phrase of the form $\langle d_q, \ell \rangle$, where $\ell \in (\ell_p, \ell_q)$ is an ℓ cost class boundary $M(v)$ for some cost class v. Thus, given all d-maximal edges, generating ℓ-maximal edges is as simple as listing all ℓ cost classes.

Ferragina, Nitto, and Venturini [20] employ two main ideas for the efficient computation of d-maximal edges outgoing from every node i:

1. *Batch processing.* The first idea allows us to bound the working space requirements to optimal $\mathcal{O}(n)$ words. It consists of executing Q passes over the graph $\mathcal{G}$, one per cost class. During the kth pass, nodes of $\mathcal{G}$ are logically partitioned into blocks of $E(k)$ contiguous nodes. Each time the computation crosses a block boundary, all the d-maximal edges spreading from that block whose copy-distance can be encoded with $L(k)$ bits are generated. These edges are kept in memory until they are used by the shortest path algorithm, and discarded as soon as the first node of the next block needs to be processed; the next block of $E(k)$ nodes is then fetched and the process repeats. Actually, all Q passes are executed in parallel to guarantee that all d-maximal edges of a node are available to the shortest path computation.
2. *Output-sensitive d-maximal generation.* The second idea allows us to compute, for each cost class k, the d-maximal edges taking $L(k)$ bits of a block of $E(k)$ contiguous nodes in $\mathcal{O}(E(k))$ time and space. As a result, the working space complexity at any instant is $\sum_{k=1}^{Q} E(k) = n$, whereas the time complexity over all Q passes is $\sum_{k=1}^{Q} \left(\frac{n}{E(k)} \, \mathcal{O}(E(k)) \right) = \mathcal{O}(n \cdot Q)$, that is, $\mathcal{O}(1)$ amortized time per d-maximal edge.

Let us now describe how to perform *batch processing*, that is, how to compute all d-maximal edges whose copy-distance can be encoded in $L(k)$ bits and which spur from

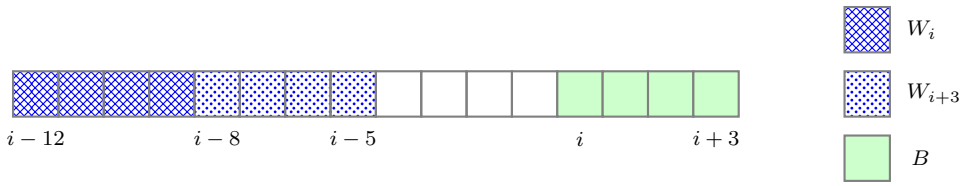


FIG. 3.1. *An illustration of W_i, W_B, and B-blocks for a cost class k with $M(k-1) = 8$, $M(k) = 12$, and $E(k) = 4$. Here, blue cells represent positions in W-blocks, while green cells represent positions in B-blocks. In this example, $W_B = W_i \cup W_{i+3}$. (Color available online only.)*

a block of nodes $B = [j, j + E(k) - 1]$. Let us consider one of these d-maximal edges $(i, i + \ell)$. Because of its d-maximality, $\mathcal{S}[i, i + \ell - 1]$ is the longest substring starting at i and having a copy at distance within $[M(k-1)+1, M(k)]$; the subsequent edge $(i, i + \ell + 1)$, if it exists, denotes a longer substring whose copy occurs in a subrange that lies farther back. Let $W_i = [i - M(k), i - M(k-1) - 1]$ be the window of nodes where the copy associated to the edge spurring from $i \in B$ must start. Since this computation must be done for all the nodes in B, it is useful to consider the *window* $W_B = W_j \cup W_{j+E(k)}$, with $|W_B| = 2E(k)$, which merges the windows of the first and the last nodes in B and thus spans all positions that can be the (copy-)reference of a d-maximal edge spurring from B (see Figure 3.1 for an illustration of these concepts). The following fact is crucial for an efficient computation of all these d-maximal edges.

FACT 3.8. *If there exists a d-maximal edge $(i, i+\ell)$ whose distance can be encoded in $L(k)$ bits, then there is a suffix $\mathcal{S}[s, n]$ starting at a position $s \in W_i$ that shares the longest common prefix (*Lcp*) with $\mathcal{S}[i, n]$ among all suffixes starting in W_i, and this* Lcp *has length ℓ. We say that position s is a* maximal position *for node i.*

The problem is thus reduced to that of efficiently finding all maximal positions for B originating in the window W_B. This is where our algorithm differs from the previous one.

3.2. New results: A new FSG algorithm. In this section we describe our novel FSG algorithm, which is denoted as Fast-FSG in section 5 in order to distinguish it from the FSG algorithm originally described by Ferragina, Nitto, and Venturini [20]. Fact 3.8 allows us to recast the problem of finding d-maximal edges (and, hence, maximal edges) as the problem of generating maximal positions of a block of nodes $B = [i, i + E(k) - 1]$; in the following we denote by W $(= W_i \cup W_{i+E(k)})$ the window where the maximal positions of B lie. We now sketch an algorithm for the problem that makes use of the suffix array $\mathsf{Sa}[1, n]$ and the inverse suffix array $\mathsf{Isa}[1, n]$ of $\mathcal{S}$. Recall that the ith entry of Sa stores the ith lexicographically smallest suffix of $\mathcal{S}$, while the ith entry of Isa is the (lexicographic) rank of the suffix $\mathcal{S}[i, n]$ (hence, $i = \mathsf{Sa}[\mathsf{Isa}[i]]$); both of these arrays can be computed in linear time [25]. Fact 3.8 implies that the d-maximal edge of a position $i \in B$ can be identified by first determining the lexicographic predecessor and successor of suffix $\mathcal{S}[i, n]$ among the suffixes starting in W_i and then taking the one with Lcp with $\mathcal{S}[i, n]$.[3] Next we show that detecting the lexicographic predecessor/successor of $\mathcal{S}[i, n]$ in W_i only requires us to scan, in lexicographic order, the set of suffixes starting in $B \cup W$. This computes the maximal position for i in constant amortized time, provided that the lexicographically sorted

[3]The Lcp can be computed in $\mathcal{O}(1)$ time with appropriate data structures, built in $\mathcal{O}(n)$ time and space [6, 25].

list of suffixes starting in $B \cup W$ is available.

We also address the crucial task of generating on-the-fly these lexicographically sorted lists taking $\mathcal{O}(1)$ time per suffix. More precisely, given a range of positions $[a, b]$, we are interested in computing the so-called *restricted suffix array* (Rsa), an array of pointers to every suffix starting in that range, arranged in lexicographic order. Consequently, Rsa is a subsequence of Sa restricted to suffixes in $[a, b]$. The key issue is to build a proper data structure that, given $[a, b]$ at query time, returns its Rsa in $\mathcal{O}(b - a + 1)$ optimal time, independent of n. We propose two solutions addressing this problem. The first, more general but less practical, is based on the *sorted range reporting* data structure [9]. The second solution is more practical and works under the assumption that, for any k, $E(k + 1)/E(k)$ is a positive integer, a condition satisfied by the vast majority of integer encoders in use, such as Elias' Gamma and Delta codes, byte-aligned codes, or (S, C)-dense codes [8, 40, 43].

Computing maximal positions. Given Fact 3.8, finding the d-maximal position of $i \in B$ (if any) can be solved by first determining the lexicographic predecessor/successor of suffix $\mathcal{S}[i, n]$ among the suffixes starting in W_i and then taking the one with Lcp with $\mathcal{S}[i, n]$. Instead of solving directly this problem, we solve a *relaxed variant* that asks for finding the lexicographic predecessor/successor of i in an *enlarged window* R_i defined as the positions in $W \cup B$ that may be copy-reference for i and can be encoded with *no more than* $L(k)$ bits (notice that, by definition, $W_j \subseteq R_j$). This actually means that we want to identify the Lcp of $\mathcal{S}[i, n]$ shared with a suffix starting at a position within the block $R_i = (W \cup B) \cap [i - M(k), i - 1]$; this is called the *relaxed maximal position* of i. The following fact shows that solving this variant suffices for our aims.

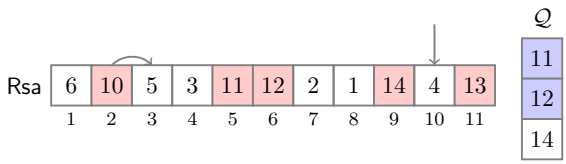
FACT 3.9. *If there exists a d-maximal edge of cost $L(k)$ bits spurring from $i \in B$, then its relaxed maximal position is a maximal position for i.*

Proof. Let us assume to the contrary that there exists a d-maximal edge (i, k) of cost $L(k)$ bits whose relaxed maximal position x is not a maximal position for this edge. By definition, $x \in B$ and thus $x \notin W_i$. This implies that the copy-distance $i - x \leq M(k - 1)$, and thus this distance is encoded with less than $L(k)$ bits. Consequently, the edge (i, k) would have a cost less than $L(k)$ bits, which is a contradiction. ∎

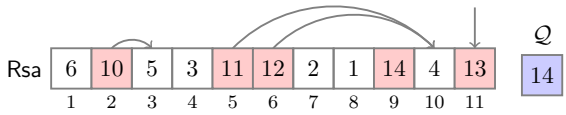
The algorithmic advantage of these enlarged windows stems from a useful *nesting property*. By definition, R_i can be decomposed as $W'_i B'_i$, where W'_i is the prefix of R_i up to the beginning of B (and it is prefixed by W_i), and B'_i is the suffix of R_i that spans from the beginning of B to position $i - 1$ (hence it is a prefix of B). The useful property is that for any $h > i$, B'_h includes B'_i (being the latter of its prefixes) and W'_h is included in W'_i (being the former of its suffixes). This is deployed algorithmically in the next paragraphs.

We solve the relaxed variant in *batch* over all suffixes starting in B in constant amortized time per suffix, thus $\mathcal{O}(|B|) = E(k)$ time per block. Our Fast-FSG solution, unlike FSG [20], is based on lists and their scans, thus is very fast in practice; moreover, like FSG, this solution is asymptotically optimal in time and space. We now focus on computing the lexicographic successors; predecessors can be found in a symmetric way.

The main idea is to scan the Rsa of $B \cup W$ rightward (hence for lexicographically increasing suffixes), keeping a queue $\mathcal{Q}$ that tracks all the suffixes starting in B but whose (lexicographic) successors in R_i have not been found yet. The queue is sorted



(a) Processing Rsa[10] which belongs to W



(b) Processing Rsa[11] which belongs to B

FIG. 3.2. *Depicted on the top is a prefix of* Rsa *yet to be processed, while on the bottom there is the corresponding queue* $\mathcal{Q}$. *In* Rsa, *red elements belong to* B, *while white elements belong to* W. *In this example,* $W = [1, 6]$, $B = [9, 13]$, *and* $M(k) = 8$. *Blue elements represent elements of* $\mathcal{Q}$ *that will be matched with the element of* Rsa *under consideration and then subsequently removed. Straight arrows indicate the element of* Rsa *to be processed, while bent arrows represent elements of* B *matched thus far. (Color available online only.)*

by starting position of its suffixes, and it is initially empty. We remark that we are reversing the problem, in that we do not ask which suffix in R_i is the successor of $\mathcal{S}[i, n]$ for $i \in B$; rather, we ask which suffix in B has the currently examined suffix $\mathcal{S}[i, n]$ of Rsa as its successor. Given the rightward (lexicographic) scan of the Rsa, the current suffix $\mathcal{S}[i, n]$ starts either in B or in W, and it is the successor of all suffixes $\mathcal{S}[j, n]$ that belong to $\mathcal{Q}$ (because they have been already scanned, and thus they are to the left of this suffix in the sorted Rsa) and their enlarged window R_j includes i. By definition of $\mathcal{Q}$, these suffixes are in B, still miss their matching successor, and are sorted by position. Hence, the rightward scan of Rsa ensures that all those $\mathcal{S}[j, n]$ have $\mathcal{S}[i, n]$ as their successor. Due to efficiency reasons, we need to find those j's without inspecting the whole queue; for this we deploy the previously mentioned nesting property as follows.

There are two cases for the currently examined suffix $\mathcal{S}[i, n]$, as depicted in Figure 3.2:

1. If $i \in B$ (Figure 3.2b), the matching suffix $\mathcal{S}[j, n]$ can be found by scanning $\mathcal{Q}$ bottom-up (i.e., backward from the largest position in $\mathcal{Q}$) and checking $i \in R_j$. This is correct, since (i) $\mathcal{Q}$ is sorted, so position j moves leftward in B and thus it does the right extreme of both R_j and its part B'_j, and (ii) $E(k) < M(k)$, so even the rightmost position in B may copy-reference any position in B with not more than $L(k)$ bits. We can then guarantee that as soon as we find the first position $j \in \mathcal{Q}$ such that $i \notin R_j$ (i.e., $i \notin B'_j$), then all the remaining positions in $\mathcal{Q}$ (which are to the left of j) will also not belong to R_j (which is moving leftward with its right extreme) and thus the scanning of $\mathcal{Q}$ can be stopped. Since all the scanned suffixes in $\mathcal{Q}$ have been matched with their successor $\mathcal{S}[i, n]$, they can be removed from the queue; moreover, since $i \in B$, it is added to the end of $\mathcal{Q}$. Notice that this last insertion keeps $\mathcal{Q}$ sorted by starting position.
2. If instead $i \in W_B$ (Figure 3.2a), all matching suffixes $\mathcal{S}[j, n]$ can be found by scanning $\mathcal{Q}$ top-down (i.e., from the smallest position) and checking whether $i \in R_j$ (i.e., $i \in W'_j$) or not. In this case the left extreme of R_j (thus the one of W'_j) moves rightward; consequently, whenever $i \notin R_j$ all positions to the right

of j in $\mathcal{Q}$ will not include i in their relaxed window, and thus the scanning of the queue can be stopped. In this case we do not add i to $\mathcal{Q}$ because $i \in W_B$, so $i \notin B$.

It is apparent that the time complexity is proportional to the number of examined/discarded elements in/from $\mathcal{Q}$, which is $|B|$, plus the cost of scanning $\mathsf{Rsa}(W \cup B)$, which is $|W| + |B|$. We have thus proved the following lemma.

LEMMA 3.10. *There exists an algorithm taking $\mathcal{O}(|W| + |B|) = E(k)$ time and space for computing the relaxed maximal positions for all positions in B.*

Restricted suffix array construction. In this section we discuss the last missing part of our algorithmic solution, namely how to construct Rsa of the set of suffixes starting in $B \cup W$, taking $\mathcal{O}(|B| + |W|)$ time and space, and thus $\mathcal{O}(n)$ space and $\mathcal{O}(n\,Q)$ time over all constructions required by the shortest path computation. Deriving on-the-fly $\mathsf{Rsa}(B \cup W)$ boils down to the problem of constructing $\mathsf{Rsa}(B)$ and $\mathsf{Rsa}(W)$ and then merging these two sorted arrays. The latter task is easy: in fact, merging the two arrays requires $\mathcal{O}(|B| + |W|)$ time by using the classic linear-time merging procedure of merge sort and by exploiting the array Isa to compare in constant time two suffixes starting at $i \in \mathsf{Rsa}(B)$ and $j \in \mathsf{Rsa}(W)$ (namely, we compare their lexicographic ranks $\mathsf{Isa}[i]$ and $\mathsf{Isa}[j]$). The linear-time construction of $\mathsf{Rsa}(B)$ and $\mathsf{Rsa}(W)$ is the hardest part. We now present two solutions. The first one, more general but less practical, is based on a slight adaptation of the *sorted range reporting* data structure by Brodal et al. [9] and achieves an optimal $\Theta(|B|+|W|)$ construction time. The second solution has the same time complexity but works under the assumption that, for any k, $E(k)$ strictly divides $E(k+1)$. The nice feature of this second solution is that, unlike the sorted range reporting data structure just mentioned, it constructs the Rsa on-the-fly avoiding any radix-sort operations, which are theoretically efficient but slow in practice.

Solution based on sorted range reporting. The sorted range reporting problem takes in input an array $A[1, n]$ to be preprocessed and asks to build a data structure that can efficiently report in sorted order the elements of any subarray $A[a, b]$ provided online. Brodal et al. solved such queries in optimal $\mathcal{O}(b - a + 1)$ time by preprocessing A in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ words of auxiliary space.

The construction of $\mathsf{Rsa}(B)$ and $\mathsf{Rsa}(W)$ reduces to this problem. In fact, assume that the pairs $\langle i, \mathsf{Isa}[i]\rangle$ for the whole range of suffixes $i \in [1, n]$ have been constructed: given a range B (resp., W), the construction of $\mathsf{Rsa}(B)$ (resp., $\mathsf{Rsa}(W)$) consists of sorting the contiguous range B (resp., W) of those pairs according to their second component, and then taking the first component of the sorted pairs.

Therefore constructing $\mathsf{Rsa}(B \cup W)$ takes $\mathcal{O}(|B| + |W| + \mathsf{Rss}(|B \cup W|))$ time where $\mathsf{Rss}(z)$ is the cost of answering a sorted range reporting query over a range of z items. Brodal et al. [9] show that this reporting takes linear time, and thus our aimed query bound is satisfied. However, constructing the data structure to support the sorted range reporting problem takes $\mathcal{O}(n \log n)$ time, which makes this solution optimal only when $Q = \Omega(\log n)$. This is the common case in practice.

The second solution to this problem, which we propose below, achieves optimality over all of Q's values by introducing a (practically irrelevant) assumption satisfied by the vast majority of integer encoders in use, such as Elias' Gamma and Delta codes, byte-aligned codes, or (S, C)-dense codes [8, 40, 43].

Scan-based, practical solution. Our second solution to the sorted range reporting problem avoids the radix-sort step, which is slow in practice, building instead

on the following intuition. Given the Rsa of a block $[a, b]$ (be it a B- or a W-block of size $E(k)$), the Rsas of evenly sized smaller blocks that are completely contained in it (such as B- and W-blocks of size $E(k-1)$) can be constructed in time $\mathcal{O}(b-a)$ by performing a simple left-to-right scan on the parent Rsa, appropriately distributing every element to one of the smaller Rsas. Under the assumption that $E(k)$ is a multiple of $E(k-1)$, every W- and B-block of length $E(k-1)$ is entirely contained within a block of size $E(k)$. Thus, every Rsa is constructed by applying this left-to-right distribution scan procedure recursively. In order to save space and keep the memory consumption inside the $\mathcal{O}(n)$ bound, an Rsa is computed only when it is needed by the shortest path algorithm, and then discarded afterwards.

In the rest of this section we illustrate the algorithmic details of this approach to prove the following lemma.

LEMMA 3.11. *Assume that, for any $k = 2, \ldots, Q$, $E(k-1)$ divides $E(k)$ and $E(k)/E(k-1) \geq 2$. There is an algorithm that generates the Rsa of any B-block and W-block in time proportional to their length, using $\mathcal{O}(n\,Q)$ preprocessing time and $\mathcal{O}(n)$ words of auxiliary space and performing only left-to-right scans of appropriate integer sequences.*

We describe only the solution for generating the Rsa of W-blocks, as the generation of the Rsa of B-blocks is analogous. Rsas of W-blocks are logically divided into Q *levels*, where level k is the set of all the Rsas of W-blocks of size $E(k)$. Let $W(k, i)$ denote the W-block of level k starting from position $(i-1)E(k)$. Since we assumed that $E(k-1)$ divides $E(k)$ for each k, every block $W(k, i)$ is completely contained within some block $W(k+1, j)$. In the following we call $W(k+1, j)$ the *father* of block $W(k, i)$, and the latter its *child*. If two blocks have the same father, then they are *siblings*.

The algorithm generates Rsas as follows. First, all Rsas for level Q are computed by means of a single distribution of the entire Sa. Then, Rsas of lower levels are generated by distributing the Rsa of their father and so on, recursively, down to level 1. The algorithm is efficient, taking optimal $\mathcal{O}(n\,Q)$ time since each Rsa is distributed exactly once. However, precomputing all Rsas at once is wasteful, as it requires $\mathcal{O}(n\,Q)$ words of memory. To overcome this space inefficiency, Rsas are computed only when needed, and discarded as soon as they are no longer required by the shortest path algorithm. The Rsa of a W-block is needed in two different moments: (i) when it is used to identify the maximal positions of a B-block, and (ii) when it is used to compute the Rsa of a lower-level W-block. Let $p = (i-1)E(k)$ be the starting position of block $W(k, i)$. It thus follows that the Rsa of a block $W(k, i)$ is first needed when processing node $p+1$ (because the leftmost W-block of level 1 is needed to compute the maximal positions of the overlapping B-block), and needed last when processing node $p + M(k)$ (when the maximal positions of the last B-block that can copy-reference positions in $W(k, i)$ are requested). The algorithm keeps, for each level k, $E(k+1)/E(k) + 1$ Rsas for W-blocks of that level. Notice that $E(k+1) \geq M(k)$, as it can be easily proved by induction. At the beginning, the Rsas of the leftmost $E(k+1)/E(k)$ W-blocks of level k are generated by means of a single, recursive, top-down generation by computing, at each level, the children of the leftmost W-block. This is enough to generate the edges outgoing from the leftmost nodes in $\mathcal{G}$. When the Rsa of a W-block is needed and not already calculated, then that block is the leftmost among its siblings: the algorithm then generates that block and all of its $E(k+1)/E(k) - 1$ siblings, overwriting all the previously computed Rsas but the rightmost one. By following this approach the algorithm uses a linear number

of words. In fact since for each $k = 1, \ldots, Q$ there are exactly $1 + E(k+1)/E(k)$ Rsas for the W-blocks of the kth cost class, the total number of words allocated for storing these Rsas is bounded by $\sum_{i=1}^{Q} \mathcal{O}\Big(\Big(1 + \frac{E(k+1)}{E(k)}\Big)\, E(k)\Big) = \mathcal{O}(n)$.

4. On the bicriteria compression. We can finally illustrate our solution for the BDCP. In the same vein as in the BLPP, we model this problem as an optimization problem on $\mathcal{G}$. More precisely, we extend the model given in section 3 by simply attaching a *time cost* $t(i,j)$ on every edge $(i,j) \in \mathcal{G}$; hence, $t(\pi) = \sum_{(i,j)\in\pi} t(i,j)$ is the decompression time of path/parsing π, and BDCP is thus reduced to the following weight-constrained shortest path problem (WCSPP):

$$\text{(WCSPP)} \qquad \min\big\{\, s(\pi) \;:\; t(\pi) \le \mathcal{T}, \quad \pi \in \Pi \,\big\},$$

where Π is the set of all $1n$-paths in $\mathcal{G}$ (or, equivalently, in $\widetilde{\mathcal{G}}$). Let $t_{\max}$ and $s_{\max}$ be, respectively, the maximum time cost and the maximum space cost of the edges in $\mathcal{G}$. Also, let $z^\star$ be the optimal solution to (WCSPP). This section proves the following theorem.

THEOREM 4.1. *There is an algorithm that computes a path π such that $s(\pi) \le z^\star + s_{\max}$ and $t(\pi) \le \mathcal{T} + 2\,t_{\max}$ in $\mathcal{O}(n\ \log n\ \log(n\ t_{\max}\ s_{\max}))$ time and $\mathcal{O}(n)$ space.*

We call this type of result an $(s_{\max}, 2\,t_{\max})$-additive approximation, which is a strong notion because the absolute error stays constant as the value of the optimal solution grows, conversely to what occurs, e.g., for the "classic" (α,β)-approximation [34] where, as the optimum grows, the absolute error grows too.

As we are using universal integer encoders and memory hierarchies that grow logarithmically, $s_{\max}$, $t_{\max} \in \mathcal{O}(\log n)$. Thus, the following corollary holds.

COROLLARY 4.2. *There is an algorithm that computes an $(\mathcal{O}(\log n), \mathcal{O}(\log n))$-additive approximation of BDCP in $\mathcal{O}\big(n\ \log^2 n\big)$ time and $\mathcal{O}(n)$ space.*

Interestingly, from Theorem 4.1 and our assumptions on $s_{\max}$ and $t_{\max}$ we can derive a fully polynomial time approximation scheme (FPTAS) for our problem as stated in the following theorem (see the proof in the appendix).

THEOREM 4.3. *For any fixed $\epsilon > 0$ there exists a multiplicative $\big(\epsilon, \frac{\epsilon}{2}\big)$-approximation scheme for the BDCP that takes $\mathcal{O}\big(\frac{1}{\epsilon}\big(n\ \log^2 n + \frac{1}{\epsilon^2}\log^4 n\big)\big)$ time and $\mathcal{O}\big(n + \frac{1}{\epsilon^3}\log^4 n\big)$ space complexity.*

By setting $\epsilon > \sqrt[3]{(\log^4 n)/n}$, the bounds become $\mathcal{O}\big(n\ \log^2 n/\epsilon\big)$ time and $\mathcal{O}(n)$ space. Notice that both the FPTAS and the (α,β)-approximation guarantee to solve the BDCP in $o(n^2)$ time complexity, as desired.

4.1. Overview of the algorithm. We now illustrate our $(s_{\max}, 2t_{\max})$ additive approximation algorithm for (WCSPP). We denote with $z(P)$ the optimal value of an optimization problem P, so that, e.g., $z^\star = z(\text{WCSPP})$, and define the *Lagrangian relaxation* of WCSPP with Lagrangian multiplier λ

$$\text{(WCSPP}(\lambda)\text{)} \qquad \min\big\{\, s(\pi) + \lambda(t(\pi) - \mathcal{T}) \;:\; \pi \in \Pi \,\big\}.$$

The algorithm works in two phases. In the first phase, described in section 4.2, the algorithm solves the *Lagrangian dual*

$$\text{(DWCSPP)} \qquad \max\big\{\, \varphi(\lambda) = z(\text{WCSPP}(\lambda)) \;:\; \lambda \ge 0 \,\big\}$$

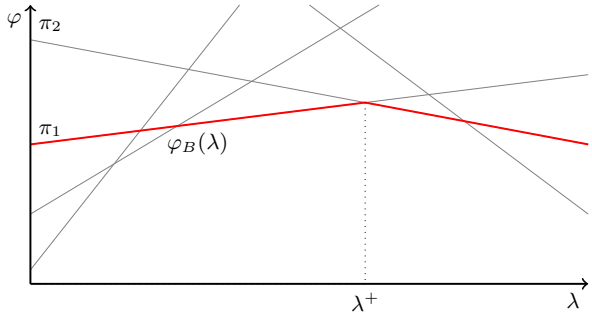


FIG. 4.1. *Each path $\pi \in B$ is a line $\varphi = L(\pi, \lambda)$, and $\varphi_B(\lambda)$ (in red) is given by the lower envelope of lines π_1 and π_2. In general $\varphi_B \geq \varphi$; in this example the maximizer of φ_B, λ^+, is the optimal solution $\lambda^\star$. (Color available online only.)*

through a specialization of Kelley's cutting-plane algorithm [28], as introduced by Handler and Zang [23]. The result is an optimal solution $\lambda^\star \geq 0$ of DWCSPP with the corresponding optimal value $\varphi^\star = \varphi(\lambda^\star) = z(\mathsf{DWCSPP})$, which is well known to provide a lower bound for WCSPP (i.e., $\varphi^\star \leq z^\star$). In addition, this computes in almost linear time (Lemma 4.4) a pair of paths (π_L, π_R) that are optimal for $\mathsf{WCSPP}(\lambda^\star)$ and such that $t(\pi_L) \geq \mathcal{T}$ and $t(\pi_R) \leq \mathcal{T}$. In case one path among them satisfies the time bound $\mathcal{T}$ exactly, then its space cost equals the optimal value $z^\star$, and thus it is an optimal solution for WCSPP. Otherwise the algorithm starts the second phase, described in section 4.3, where, exploiting the specific properties of our graph $\mathcal{G}$ and of the coefficients of functions $s(-)$ and $t(-)$, we construct a new path by joining a proper prefix of π_L with a proper suffix of π_R. The key difficulty here is to show that this new path guarantees an additive approximation of the optimal solution (Lemma 4.11), and it can be computed in just $\mathcal{O}(n)$ time and $\mathcal{O}(1)$ auxiliary space.

4.2. First phase: The cutting-plane algorithm. The Lagrangian dual problem DWCSPP can be solved with a simple iterative process whose complexity has been studied by Handler and Zang [23]. The key of the approach is to rewrite DWCSPP as the (very large) linear program

$$\max \{ \varphi \ : \ \varphi \leq s(\pi) + \lambda(t(\pi) - \mathcal{T}), \quad \pi \in \Pi, \quad \lambda \geq 0 \},$$

in which every $1n$-path defines one of the constraints and, possibly, one face of the feasible region. This can be easily described geometrically. Let us denote as $L(\pi, \lambda) = s(\pi) + \lambda(t(\pi) - \mathcal{T})$ the Lagrangian cost, or λ-*cost*, of the path π with parameter λ. Each path π thus represents the line $\varphi = L(\pi, \lambda)$ in the Euclidean space (λ, φ): feasible paths have a nonpositive slope (since $t(\pi) \leq \mathcal{T}$), while unfeasible paths have a positive slope (since $t(\pi) > \mathcal{T}$). The *Lagrangian function* $\varphi(\lambda)$ defined in DWCSPP is then the pointwise maximum of all the lines $L(\pi, \lambda)$ for $\pi \in \Pi$; φ is therefore piecewise linear and represents the lower envelope of all the lines, as illustrated in Figure 4.1. The exponential number of paths makes it impossible to solve WCSPP by a brute-force approach; however, the full set of paths Π is not needed. In fact, we can use a *cutting-plane* method [28], which determines a pair of paths (π_L, π_R) such that (i) $L(\pi_L, \lambda^\star) = L(\pi_R, \lambda^\star) = z^\star$, and (ii) $t(\pi_L) \geq \mathcal{T}$ and $t(\pi_R) \leq \mathcal{T}$. In the context of the simplex method [39], these paths correspond to an *optimal basis* of the linear program.

At each step, the cutting-plane algorithm keeps a pair $B = (\pi_1, \pi_2)$ of $1n$-paths.

The initial pair B is given by the *space-optimal* and the *time-optimal* paths, respectively, which can be obtained by means of two shortest path computations over $\widetilde{\mathcal{G}}$. π_1 has to have a positive slope ($t(\pi_1) > \mathcal{T}$), for otherwise it is optimal and we can stop. π_2 has to have a nonpositive slope ($t(\pi_2) \leq \mathcal{T}$), for otherwise WCSPP has no feasible solution: π_2 is therefore the "least unfeasible" solution. This *feasible-unfeasible* invariant is kept true along the iterations. The set B defines the *model* $\varphi_B(\lambda)$, which is a restriction of the function $\varphi(\lambda)$ to the (two) paths in $B \subset \Pi$, as illustrated in Figure 4.1. It is also easy to see that the intersection point (λ^+, φ^+) between $L(\pi_1, \lambda)$ and $L(\pi_2, \lambda)$, which exists because the two slopes have opposite sign, corresponds to the maximum of the function $\varphi_B(\lambda)$, as illustrated in Figure 4.1. In other words, $\lambda^+ \geq 0$ maximizes φ_B over all $\lambda \geq 0$, and φ^+ is the optimal value. Since $\varphi(\lambda)$ is the lower envelope of the lines given by *all* paths in $\Pi \supseteq B$, it holds that $\varphi_B(\lambda) \geq \varphi(\lambda)$; as a corollary, $\varphi^+ \geq \varphi^\star$.

At each step, after having determined λ^+ the algorithm solves the Lagrangian relaxation WCSPP(λ^+), i.e., it computes a path π^+ for which $L(\pi^+, \lambda^+) = \varphi(\lambda^+)$. Section 3.1 shows that the path π^+ can be determined efficiently by solving a shortest path problem on the pruned DAG $\widetilde{\mathcal{G}}$. If $\varphi(\lambda^+) = \varphi^+$, then the current $B = (\pi_1, \pi_2)$ is an optimal basis, as it is immediate to verify ($\varphi(\lambda^+) \leq \varphi^\star \leq \varphi^+$), and the algorithm stops returning $\lambda^\star = \lambda^+$ and $(\pi_L, \pi_R) = B$. Otherwise (i.e., $\varphi(\lambda^+) < \varphi^+$) B is not an optimal basis, and the algorithm must update B to maintain the *feasible-unfeasible* invariant on φ_B stated above; a simple geometric argument shows that B can be updated as (π_1, π^+) if π^+ is feasible and as (π^+, π_2) if it is not.

It is crucial to estimate how many iterations the cutting-plane algorithm requires to find the optimal solution. Mehlhorn and Ziegelmann have shown [38] that, if the costs and the resources of each arc are integers belonging to the compact sets $[0, C]$ and $[0, R]$, respectively, then the cutting-plane algorithm (which they refer to as the *Hull approach*) terminates in $\mathcal{O}(\log(nRC))$ iterations. Since in our context $R = t_{\max}$ and $C = s_{\max}$, we have the following.

LEMMA 4.4. *The first phase of the Bicriteria compression algorithm computes a lower bound $\varphi^\star$ for WCSPP, an optimal solution $\lambda^\star \geq 0$ of DWCSPP, and a pair of paths (π_L, π_R) that are optimal for WCSPP($\lambda^\star$). This takes $\mathcal{O}(\tilde{m} \log(n\, t_{\max}\, s_{\max}))$ time and $\mathcal{O}(n)$ space, where the parameter $\tilde{m} = \mathcal{O}(n \log n)$ denotes the size of $\widetilde{\mathcal{G}}$.*

4.3. Second phase: The path-swapping algorithm. Unfortunately, it is not easy to bound the solution computed with Lemma 4.4 in terms of the space-optimal solution of WCSPP. Therefore, the second phase of our algorithm is the crucial step that allows us to turn the basis (π_L, π_R) into a path whose time and space costs can be bounded in terms of the optimal solution for WCSPP. The intuition here is that it is possible to combine together the optimal Lagrangian dual basis (π_L, π_R) to get a quasi-optimal solution through a *path-swap* operation. This idea has been inspired by the work of [3], where a similar *path-swap* operation has been used to solve exactly a WCSPP with unitary weights and costs that satisfy the Monge condition.

In the following, paths are nonrepeating sequences of increasing node-IDs, so that (v, w, w, w, z) must be intended as (v, w, z). Moreover, we say that

- $\mathsf{Pref}(\pi, v)$ is the *prefix* of a path π ending in the largest node $v' \leq v$ in π;
- $\mathsf{Suf}(\pi, v)$ is the *suffix* of a path π starting from the smallest node $v'' \geq v$ in π.

Given two paths π_1 and π_2 in $\mathcal{G}$, we call *path-swapping* through a *swapping-point* v that belongs to either π_1 or π_2 (or both) the operation that creates a new path, denoted by $\mathrm{ps}(\pi_1, \pi_2, v) = (\mathsf{Pref}(\pi_1, v), v, \mathsf{Suf}(\pi_2, v))$, that connects a prefix of π_1

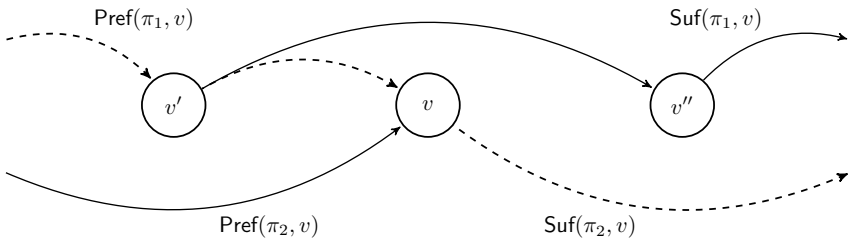


FIG. 4.2. *A path-swap of π_1, π_2 at the swapping-point v. The resulting path is dashed.*

with a suffix of π_2 via v. Fact 4.5 states that this operation is well defined. In fact, Property 3.2 guarantees that the edges connecting the last node of $\mathsf{Pref}(\pi_1, v)$ with v, and v with the first node of $\mathsf{Suf}(\pi_2, v)$ do exist. An illustrative example is provided in Figure 4.2.

FACT 4.5. *The path-swap operation is well defined for each pair (π_1, π_2) of 1n-paths and for each swapping-point v that belongs to either π_1 or π_2 (or both).*

We now illustrate some useful properties on the path-swapped paths. Let us consider an arbitrary swapping-point v and the two path-swapped solutions $\pi_A = \mathrm{ps}(\pi_1, \pi_2, v)$ and $\pi_B = \mathrm{ps}(\pi_2, \pi_1, v)$. The following lemma shows that the sums of the compressed spaces and decompression times of solutions π_A and π_B are "close" to those of π_1 and π_2.

LEMMA 4.6. *Let π_1, π_2 be any two s-t paths in $\mathcal{G}$. Let us consider the paths π_A, π_B obtained by swapping π_1 and π_2 at any arbitrary swapping-point v. Then the following holds:*

- $s(\pi_A) + s(\pi_B) \leq s(\pi_1) + s(\pi_2) + s_{\max}$,
- $t(\pi_A) + t(\pi_B) \leq t(\pi_1) + t(\pi_2) + t_{\max}$.

Proof. In the following we prove the theorem only for the space bound; the proof for the time is symmetrical. First, let us note that $s(\pi_1) = s(\mathsf{Pref}(\pi_1)) + s(\mathsf{Suf}(\pi_1))$; the same goes for π_2.

There are three cases to consider:

1. *v belongs to both π_1 and π_2*: in this case, we have $\pi_A = (\mathsf{Pref}(\pi_1, v), \mathsf{Suf}(\pi_2, v))$ and $\pi_B = (\mathsf{Pref}(\pi_2, v), \mathsf{Suf}(\pi_1, v))$, so in this case $s(\pi_A) + s(\pi_B) = s(\pi_1) + s(\pi_2)$.
2. *v does not belong to π_1*: let v' and v'' be, respectively, the rightmost node preceding v and the leftmost node following v in π_1 (see Figure 4.2). In this case, we have
   - $s(\pi_1) = s(\mathsf{Pref}(\pi_1, v)) + s(v', v'') + s(\mathsf{Suf}(\pi_1, v))$,
   - $s(\pi_2) = s(\mathsf{Pref}(\pi_2, v)) + s(\mathsf{Suf}(\pi_2, v))$,
   - $s(\pi_A) = s(\mathsf{Pref}(\pi_1, v)) + s(v', v) + s(\mathsf{Suf}(\pi_2, v))$, and
   - $s(\pi_B) = s(\mathsf{Pref}(\pi_2, v)) + s(v, v'') + s(\mathsf{Suf}(\pi_1, v))$.

   Therefore,
   - $s(\pi_1) + s(\pi_2) = s(\mathsf{Pref}(\pi_1, v)) + s(\mathsf{Pref}(\pi_2, v)) + s(\mathsf{Suf}(\pi_1, v)) + s(\mathsf{Suf}(\pi_2, v)) + s(v', v'')$, and
   - $s(\pi_A) + s(\pi_B) = s(\mathsf{Pref}(\pi_1, v)) + s(\mathsf{Pref}(\pi_2, v)) + s(\mathsf{Suf}(\pi_1, v)) + s(\mathsf{Suf}(\pi_2, v)) + s(v', v) + s(v, v'')$.

   Since $s(v', v) \leq s(v', v'')$, it easily follows that $s(\pi_A) + s(\pi_B) \leq s(\pi_1) + s(\pi_2) + s(v, v'') \leq s(\pi_1) + s(\pi_2) + s_{\max}$.
3. *v does not belong to π_2*: this case is symmetrical to the previous one. ∎

For any given $\lambda \geq 0$, a path π is λ-*optimal* if its Lagrangian cost $L(\pi, \lambda)$ is equal to the value of the Lagrangian function $\varphi(\lambda)$. It is useful to display λ-optimality geometrically. Let us consider the bidimensional space where a point (s, t) denotes a solution with compressed space s and decompression time t; with this interpretation, a path π is λ-optimal if its coordinates $(s(\pi), t(\pi))$ lie in the line $s + \lambda(t - \mathcal{T}) = \varphi$, which has a negative slope. Moreover, any path π must have its (s, t)-coordinates lying above that line. The following simple geometrical lemma implies that the $(s,\ t)$-coordinates of a path π are "far" from any λ-optimal point in either space or time *if and only if* there exists a point $(s',\ t')$ for which π is "far" in *both* components.

LEMMA 4.7. *Let $y = f(x) = b - ax$ be a line in a bidimensional space for some constants $a > 0$ and b. Let also $p = (x^\star, y^\star)$ be a point in that bidimensional space. The following statements are equivalent:*

1. *Point p lies below $y = f(x)$, that is, $y^\star < f(x^\star)$.*
2. *For any point $p = (x', y')$ such that $y' = f(x')$, either $x^\star < x'$ or $y^\star < y'$ or both.*
3. *There exists a point $p = (x', y')$ such that $y' = f(x')$ and both $x^\star < x'$ and $y^\star < y'$ (that is, p* dominates *$(x^\star, y^\star)$).*

Proof. We now show their equivalence through a circular chain of implications.

(1 → 2) Let (x', y') be such that $y' + ax' = b$. Let us assume by contradiction that $x' < x^\star$ and $y' < y^\star$; then $y^\star + ax^\star > y' + ax' = b$, which is absurd.

(2 → 3) Let $(x', y^\star)$ with $x' = f(y^\star)$; from statement 2, it holds that $x' > x^\star$ and so it immediately follows from the definition of $f(x)$ that, for each x in the range $(x^\star, x')$, it holds that both $x > x^\star$ and $f(x) > y^\star$.

(3 → 1) From $y' + ax' = b$, $x^\star < x'$, and $y^\star < y'$ it follows that $y^\star + ax^\star > b$. ∎

The following lemma shows that any path-swap of two λ-optimal paths is off at most $t_{\max}$ in time and $s_{\max}$ in space from being a λ-optimal path. It shows that if π_A is off by more than $t_{\max}$ and $s_{\max}$ from some λ-optimal points in the (s, c)-space, then its λ-cost is exceedingly high and, by Lemma 4.6, π_B would have a λ-cost lower than φ. By Lemma 4.7, from this impossibility result follows the existence of λ-optimal points from which π is within $s_{\max}$ and $t_{\max}$ in space and time.

LEMMA 4.8. *Let π_1, π_2 be λ-optimal paths for some $\lambda \geq 0$. Consider the path $\pi_A = \mathrm{ps}(\pi_1, \pi_2, v)$, where v is an arbitrary swapping-point: there exist values s, t such that $s(\pi_A) \leq s + s_{\max}$, $t(\pi_A) \leq t + t_{\max}$, and $s + \lambda(t - \mathcal{T}) = \varphi(\lambda)$.*

Proof. Let $S = s(\pi_1) + s(\pi_2)$ and $T = t(\pi_1) + t(\pi_2)$; since both π_1 and π_2 are λ-optimal solutions, it follows quite easily that $S + \lambda(T - 2\,\mathcal{T}) = 2\varphi$. Let also $S' = s(\pi_A) + s(\pi_B)$ and $T' = t(\pi_A) + t(\pi_B)$. Due to Lemma 4.6, it follows that $S' + \lambda(T' - 2\,\mathcal{T}) \leq 2\,\varphi + s_{\max} + \lambda t_{\max}$. Now let us suppose to the contrary that there is a couple $(s,\ t)$ such that $s + \lambda(t - \mathcal{T}) = \varphi(\lambda)$, $s(\pi_A) > s + s_{\max}$, and $t(\pi_A) > t + t_{\max}$. In particular, this implies that $s(\pi_A) + \lambda(t(\pi_A) - \mathcal{T}) > \varphi + s_{\max} + \lambda t_{\max}$, and so we have

$$\begin{aligned} s(\pi_B) + \lambda(t(\pi_B) - \mathcal{T}) &= S' + \lambda(T' - 2\mathcal{T}) - (s(\pi_A) + \lambda(t(\pi_A) - \mathcal{T})) \\ &< (2\,\varphi + s_{\max} + \lambda t_{\max}) - (\varphi + s_{\max} + \lambda t_{\max}) \\ &= \varphi, \end{aligned}$$

which is absurd because it violates the assumption on the λ-optimality of π_1 and π_2.

Let us now interpret this result in a geometric fashion. Let us consider the bidimensional space defined by the decompression time/compressed space coordinates.

The set L of points (s,t) satisfying the equation $s+\lambda(t-\mathcal{T})=\varphi(\lambda)$ is thus a line in such space. Let us now consider the point $p^\star$ with space coordinate $s(\pi_A)-s_{\max}$ and time coordinate $t(\pi_A)-t_{\max}$. In this interpretation, it thus holds that for each point $(s,\,t)\in L$, the point $p^\star$ is such that either $s(\pi_A)-s_{\max}<s$ or $t(\pi_A)-t_{\max}<t$. By Lemma 4.7, this implies that point $p^\star$ is *below* line L, so there is a point $(s',p')\in L$ such that $s'>s(\pi_A)-s_{\max}$ and $t'>t(\pi_A)-t_{\max}$. ∎

Now, consider two paths π_1, π_2 to be swapped and two *consecutive swapping-points*, that is, two nodes v and w belonging to either π_1 or π_2 and such that there is no node z belonging to π_1 or π_2 with $v<z<w$. Lemma 4.9 states that the time and space costs of paths $\mathrm{ps}\,(\pi_1,\pi_2,v)$ and $\mathrm{ps}\,(\pi_1,\pi_2,w)$ differ by at most $t_{\max}$ and $s_{\max}$, respectively.

LEMMA 4.9. *Let π_1, π_2 be two paths to be swapped. Let also v and w be two consecutive swapping points. Set $\pi=\mathrm{ps}\,(\pi_1,\pi_2,v)$ and $\pi'=\mathrm{ps}\,(\pi_1,\pi_2,w)$; then, $|s(\pi)-s(\pi')|\le s_{\max}$ and $|t(\pi)-t(\pi')|\le t_{\max}$.*

Proof. Let us consider the subpaths $\mathsf{Pref}=\mathsf{Pref}(\pi,v)$ and $\mathsf{Pref}'=\mathsf{Pref}(\pi',w)$. There are two cases:

1. $v\in\pi_1$: in this case, $\mathsf{Pref}'=(\mathsf{Pref},w)$. Thus, $s(\mathsf{Pref}')-s(\mathsf{Pref})=s(v,w)$ and $t(\mathsf{Pref}')-t(\mathsf{Pref})=t(v,w)$.
2. $v\notin\pi_1$: let $\mathsf{Pref}=(v_1,\dots,v_k,v)$; in this case, we have $\mathsf{Pref}'=(v_1,\dots,v_k,w)$. Thus, we have $s(\mathsf{Pref}')-s(\mathsf{Pref})=s(v_k,w)-s(v_k,v)\le s_{\max}$; a similar argument holds for the time cost.

Thus, $s(\mathsf{Pref}')-s(\mathsf{Pref})\le s_{\max}$ and $t(\mathsf{Pref}')-t(\mathsf{Pref})\le t_{\max}$. Symmetrically, it holds that $s(\mathsf{Suf})-s(\mathsf{Suf}')\le s_{\max}$ and $t(\mathsf{Suf})-t(\mathsf{Suf}')\le t_{\max}$; since $s(\pi)=s(\mathsf{Pref})+s(\mathsf{Suf})$ and $s(\pi')=s(\mathsf{Pref}')+s(\mathsf{Suf}')$, it follows that $|s(\pi)-s(\pi')|\le s_{\max}$, and a similar argument holds for $|t(\pi)-t(\pi')|$. ∎

Figure 4.3 gives a geometrical interpretation of this lemma and shows, in an intuitive way, that it is possible to path-swap the optimal basis (π_L,π_R) computed by the cutting-plane algorithm (Lemma 4.4) to get an additive $(s_{\max},2\,t_{\max})$-approximation to (WCSPP) by picking any path-swapped solution π with decompression time bounded between $\mathcal{T}+t_{\max}$ and $\mathcal{T}+2\,t_{\max}$, as shown in the following lemma. In this way, since $t(\pi)\ge\mathcal{T}+t_{\max}$, path π is at most $t_{\max}$ away from being a λ-optimal solution with decompression time $\ge\mathcal{T}$. Since λ-optimal solutions with decompression time higher than $\mathcal{T}$ have smaller compression space, the result readily follows.

LEMMA 4.10. *Given an optimal basis (π_L,π_R) with $t(\pi_L)\ge\mathcal{T}$ and $t(\pi_R)\le\mathcal{T}$, there exist a swapping-point $v^\star$ and a path-swapped path $\pi^\star=\mathrm{ps}\,(\pi_1,\pi_2,v^\star)$ such that $t(\pi^\star)\le\mathcal{T}+2\,t_{\max}$ and $s(\pi^\star)\le\varphi^\star+s_{\max}$.*

Proof. Since $\mathrm{ps}\,(\pi_L,\pi_R,v_1)=\pi_R$ and $\mathrm{ps}\,(\pi_L,\pi_R,v_n)=\pi_L$, Lemma 4.9 implies that there must exist some $v^\star$ such that the path $\pi^\star=\mathrm{ps}\,(\pi_L,\pi_R,v^\star)$ has time $t(\pi^\star)\in[\,\mathcal{T}+t_{\max}\,,\,\mathcal{T}+2\,t_{\max}\,]$. Due to Lemma 4.8, there are $s\ge s(\pi^\star)-s_{\max}$ and $t\ge\mathcal{T}$ (since $t+t_{\max}\ge t(\pi^\star)\ge\mathcal{T}+t_{\max}$) such that $s+\lambda(t-\mathcal{T})=\varphi^\star$; hence $s\le\varphi^\star$, which ultimately yields that $s(\pi^\star)\le\varphi^\star+s_{\max}$. ∎

The gap-closing procedure thus consists of choosing the best path-swap of the optimal basis (π_L,π_R) with time cost within $\mathcal{T}+2\,t_{\max}$. The solution can be selected by scanning left-to-right all the swapping-points and evaluating the time cost and space cost for each candidate. This can be efficiently implemented by keeping the time and space costs of the current prefix of π_L and suffix of π_R, and by updating them every time a new swapping-point is considered. Since each update can be performed

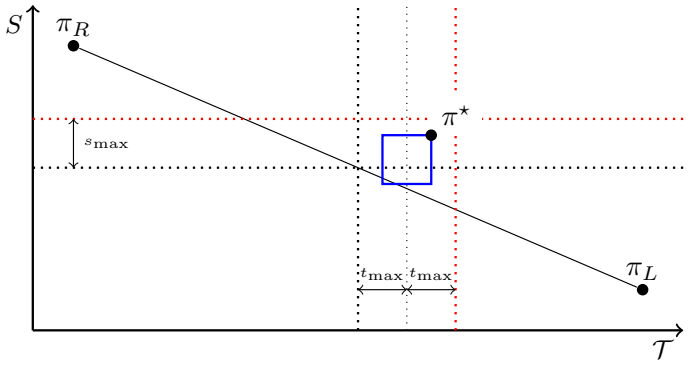


FIG. 4.3. *Geometrical interpretation of Lemmas* 4.8 *and* 4.9. *Paths are represented as points in the time/space coordinates. Path* $\pi^\star$ *is obtained by path-swapping paths* π_L *and* π_R. *The blue rectangle is guaranteed by Lemma* 4.8 *to intersect with the segment from* π_L *to* π_R, *while Lemma* 4.9 *guarantees that there is at least one path-swapped solution having time coordinates between* t *and* $t + t_{\max}$ *for any* $t \in [t(\pi_R), t(\pi_L)]$, *in this case* $[\mathcal{T} + t_{\max}, \mathcal{T} + 2t_{\max}]$. *(Color available online only.)*

in $\mathcal{O}(1)$ time, we obtain the following lemma that, combined with Lemma 4.4, proves our main theorem, Theorem 4.1.

LEMMA 4.11. *Given an optimal basis* (π_L, π_R) *of problem* DWCSPP, *the second phase of our bicriteria-compression algorithm finds an additive* $(s_{\max}, 2\,t_{\max})$*-approximation to* WCSPP *taking* $\mathcal{O}(n)$ *time and* $\mathcal{O}(1)$ *auxiliary space.*

Given the results in Lemmas 4.4 and 4.11 we eventually proved Theorem 4.1.

4.4. Generalized theorems for the WCSPP. We now show that the efficient algorithm for the WCSPP problem illustrated in this section can also be applied either to the broad family of weighted DAGs that satisfy Property 3.2 and a monotonicity property on their costs and weights, with an additive guarantee similar to that in Theorem 4.1 (Theorem 4.12), or just Property 3.2, with only a slightly worse additive approximation (Theorem 4.13). Here we remark that these results are especially relevant for instances of the weight-constrained shortest path problem (WCSPP) over graphs with a large diameter and "small" costs and weights, where the additive approximations of Theorems 4.12 and 4.13 are especially good.

Let us consider the general definition of the WCSPP over a graph $\mathcal{G} = (V, E)$ where $c(e)$ and $w(e)$ are the cost and weight of an edge $e \in E$, and $c(\pi)$ and $w(\pi)$ are the cost and weight of a path π belonging to the set Π of all source-target paths in $\mathcal{G}$:

$$\min \left\{ c(\pi) \ : \ w(\pi) \leq \mathcal{W}, \quad \pi \in \Pi \right\},$$

The following theorem generalizes Lemma 4.11 by explicitly stating the set of properties of $\mathcal{G}$ that we assumed in sections 4.2 and 4.3.

THEOREM 4.12. *Let* $\mathcal{G} = (V, E)$ *be a DAG where each edge* $e \in E$ *has cost* $c(e)$ *and weight* $w(e)$. *Let also* $n = |V|$, $m = |E|$, *and let* $c_{\max}$ *and* $w_{\max}$ *be, respectively, the maximum cost and weight of any edge in* E. *Pick any topological sort of* $\mathcal{G}$ *and let* v_i *be the* i*th edge according to that order. If* $\mathcal{G}$ *satisfies the properties*

1. *if* $(v_i, v_j) \in E$ *implies* $(v_a, v_b) \in E$ *for all* $i \leq a < b \leq j$ *(Property* 3.2*),*
2. *if* $e = (v_i, v_j)$ *and* $e' = (v_a, v_b)$ *with* $i \leq a < b \leq j$, *then* $c(e) \leq c(e')$ *and* $w(e) \leq w(e')$ *(nondecreasing cost property),*

then there exists an algorithm that computes an additive $(c_{\max}, 2\cdot w_{\max})$*-approximation to* WCSPP *taking* $\mathcal{O}(m \log(n \cdot s_{max} \cdot c_{max}))$ *time and* $\mathcal{O}(1)$ *auxiliary space.*

We now observe that the nondecreasing cost property requirement can be dropped at only a slightly worse additive approximation result.

THEOREM 4.13. *Let* $\mathcal{G} = (V, E)$ *be a DAG where each edge* $e \in E$ *has cost* $c(e)$ *and weight* $w(e)$*. Let also* $n = |V|$*,* $m = |E|$*, and let* $c_{\max}$ *and* $w_{\max}$ *be, respectively, the maximum cost and weight of any edge in* E*. Pick any topological sort of* $\mathcal{G}$ *and let* v_i *be the* i*th edge according to that order. If* $\mathcal{G}$ *satisfies the property*

$$(v_i, v_j) \in E \text{ implies } (v_a, v_b) \in E \text{ for all } i \le a < b \le j \text{ (Property 3.2),}$$

then there exists an algorithm that computes an additive $(2 \cdot c_{\max}, 3 \cdot w_{\max})$*-approximation to* WCSPP *taking* $\mathcal{O}(m \log(n \cdot s_{max} \cdot c_{max}))$ *time and* $\mathcal{O}(1)$ *auxiliary space.*

This result holds because we can use the same argument used in Lemma 4.6, which is the only lemma where the nondecreasing property has been explicitly used, to prove the following lemma that does not assume that property.

LEMMA 4.14. *Let* π_1*,* π_2 *be any two* s*-*t *paths in a graph* $\mathcal{G}$ *defined as in Theorem* 4.13*. Let us consider the paths* π_A*,* π_B *obtained by swapping* π_1 *and* π_2 *at any arbitrary swapping-point* v*. Then, the following hold:*

- $c(\pi_A) + c(\pi_B) \le c(\pi_1) + c(\pi_2) + 2 \cdot c_{\max}$*;*
- $w(\pi_A) + w(\pi_B) \le w(\pi_1) + w(\pi_2) + 2 \cdot w_{\max}$*.*

Proof. The only difference with respect to Lemma 4.6 is when the swapping-point v does not belong to π_1. Let us consider only the cost inequality, as the argument for the weight inequality is analogous. In this case, we have

- $c(\pi_1) + c(\pi_2) = c(\mathsf{Pref}(\pi_1, v)) + c(\mathsf{Pref}(\pi_2, v)) + c(\mathsf{Suf}(\pi_1, v)) + c(\mathsf{Suf}(\pi_2, v)) + c(v', v'')$,
- $c(\pi_A) + c(\pi_B) = c(\mathsf{Pref}(\pi_1, v)) + c(\mathsf{Pref}(\pi_2, v)) + c(\mathsf{Suf}(\pi_1, v)) + c(\mathsf{Suf}(\pi_2, v)) + c(v', v) + c(v, v'')$.

Since $c(v', v) + c(v', v'') - c(v, v'') \le 2 \cdot c_{\max}$, it easily follows that $c(\pi_A) + c(\pi_B) \le c(\pi_1) + c(\pi_2) + 2 \cdot c_{\max}$. ∎

Analogously, the following holds by combining Lemma 4.8 and Lemma 4.14.

LEMMA 4.15. *Let* π_1*,* π_2 *be two* λ*-optimal paths for some* $\lambda \ge 0$*. Consider the path* $\pi_A = \mathrm{ps}(\pi_1, \pi_2, v)$*, where* v *is an arbitrary swapping point: there exist values* c*,* w *such that* $c(\pi_A) \le c + 2 \cdot c_{\max}$*,* $w(\pi_A) \le w + w_{\max}$*, and* $w + \lambda(w - \mathcal{W}) = \varphi(\lambda)$*.*

Finally, by using Lemmas 4.14 and 4.15 in the same argument of Lemma 4.10, we obtain Theorem 4.13.

5. Experimental results. In this section we characterize the effectiveness of the bicriteria-based approach by discussing all the crucial algorithmic components of a well-engineered implementation and its performance on real-world datasets.

First we show how to extend the bit-optimal LZ77-parsing to include also *literal strings*, that is, phrases of the form $\langle \alpha, \ell \rangle_L$ where α is a string of length ℓ, with the same time and space bounds of Theorem 3.1. Literal runs are useful when representing incompressible portions of $\mathcal{S}$ since they improve both compression ratio and decompression speed, as shown later in this section.

We then evaluate the advantage of the bit-optimal parsing against the greedy parsing, the most popular LZ77-parsing strategy, and many high-performance compressors. We show that the bit-optimal parsing has a clear advantage over heuristically highly engineered compressors, thus justifying the interest in our technology. We also

show that the novel Fast-FSG algorithm presented in section 3.2 helps to considerably reduce the compression time, thus making bit-optimal parsing a solid and practical technology.

Next, we compare bicriteria data compression against the most common approaches for trading decompression time for compression ratio. Many practical LZ77 implementations (e.g., Gzip, Snappy, Lz4) employ the *bucketing strategy*, that is, splitting the file into blocks (buckets) of equal size which are individually compressed and then concatenated to produce the compressed output. Alternatively, a *moving window* is used to (hopefully) lower decompression time by forcing spatial locality via a limit on the maximum distance at which a phrase may be copied. We show that these heuristics are not always effective to speed up file decompression, basically because they take into account neither integer decoding time nor the length of the copied string, which may be relevant in some cases and could amortize the cost of long but far copies. We validate this argument by introducing in section 5.3 a *decompression time model* that properly infers the decompression time of an LZ77-parsing from a small set of features (such as number of copies, distance distribution, etc.). This model is used to efficiently determine proper time costs for the edges of the graph over which WCSPP is solved.

Finally we show and comment on the vast time/space trade-offs achievable with the bicriteria strategy, and point out that it improves *simultaneously* on both the most succinct (like Bzip2) and the fastest (like Lz4) compressors.

Experimental setting. The compressor has been implemented in C++ and compiled with the Intel C++ Compiler 14 with flags `-O3 -DNDEBUG -march=native`. According to the applicative scenario we have in mind, namely, "compress once, decompress many times," we used two machines to carry out the experiments. The first, used in compression, is equipped with AMD Opteron 6276 processors and 128GB of memory, while the second, used in decompression, is equipped with an Intel Core i5-2500, with 8GB of DDR3 1333MHz memory. Both machines run Ubuntu 12.04.

Experiments were executed over 1GiB-long (2^{30} bytes) datasets of different types:

- Census: U.S. demographic statistics in tabular format (type: database);
- Dna: collection of families of genomes (type: highly repetitive biological data);
- Mingw: archive containing the whole MinGW software distribution (type: mix of source codes and binaries);[4]
- Wikipedia: dump of English Wikipedia (type: natural language).

Each dataset has been obtained by taking a random chunk of 1GiB from the complete files. The experimental setting (code and documentation) is available at https://github.com/farruggia/bc-zip.

5.1. Allowing literal strings. Modern and highly performing compressors such as Snappy or Lz4 use a different phrase representation when compressing incompressible portions of the input text, in order to improve both decompression speed and compression efficacy. The idea is to represent a substring α of ℓ characters via a so-called *literal string* $\langle \ell, \alpha \rangle_L$, such that the substring α is represented as is, preceded by the integer ℓ properly encoded via a variable-length encoder enc.

The issue is how to extend the shortest path computation with literal strings (and thus, literal edges) *for any possible Lagrangian relaxation*. To this end, let us first extend the time/space model illustrated in section 2.

- *Space cost.* The space cost of a literal edge $\langle \ell, \alpha \rangle_L$ is given by $C + |\mathsf{enc}(\ell)| +$

[4]Courtesy of Matt Mahoney (http://mattmahoney.net/dc/mingw.html).

$\ell \log \sigma$, where C is a constant that depends on how a copy-phrase is distinguished from a literal-phrase in the compressed representation.

- *Time cost.* The time cost is given by the time needed to unpack the literal length ℓ, taking $t_U(\ell) \propto |\mathsf{enc}(\ell)|$ time, and the time needed to copy/write the ℓ characters of α, taking $\ell\, t_{\mathcal{C}}$ time. Furthermore, since literal-phrases are much less frequent than copy-phrases, their occurrence induces a branch misprediction that we account for with an additive term t_B. Summing up, the time needed for processing a phrase $\langle \ell, \alpha \rangle_L$ is $t_B + t_U(\ell) + \ell\, t_{\mathcal{C}}$.

Both the space and time costs have a fixed part (C, t_B), a part that is proportional to $|\mathsf{enc}(\ell)|$ ($|\mathsf{enc}(\ell)|$ itself and $t_U(\ell)$), and a part that is proportional to ℓ ($\ell \log \sigma$ and $\ell t_{\mathcal{C}}$). This implies that, for every λ, the Lagrangian cost $s(\langle \ell, \alpha \rangle_L) + \lambda t(\langle \ell, \alpha \rangle_L)$ always has the same structure. In other words, the λ-cost for a literal edge $\langle \ell, \alpha \rangle_L$ has the form $\mathsf{x} + \mathsf{y}|\mathsf{enc}(\ell)| + \mathsf{z}\,\ell$.

Given these costs, one could think it is enough to construct a literal edge $(i, j)_L$ for each pair of nodes i and j in $\mathcal{G}$ (they are $\Theta(n^2)$) and then plainly adapt the pruning strategy described in section 3.1. Unfortunately this cannot be done because each literal edge outgoing from a node has a distinct and increasing cost with ℓ. Still we are able to design a simple strategy for identifying at most Q of literal edges that spur from each node.

Let us consider two literal edges $(i, h)_L$ and $(j, h)_L$ having lengths ℓ_i and ℓ_j, respectively, and both being of the kth cost class for enc. Moreover, let us denote as $c[i]$ the cost of the optimal path from node 1 to node i in $\mathcal{G}$. The algorithm will select the edge $(i, h)_L$ over the edge $(j, h)_L$ if $c[i] + \ell_i\,\mathsf{z} < c[j] + \ell_j\,\mathsf{z}$, i.e., if $c[i] < c[j] + (\ell_j - \ell_i)\mathsf{z}$. It thus follows that, for a given cost class k, the candidate literal edge (j, h) is the one minimizing $g(i) = c[i] - i\,\mathsf{z}$ among the nodes $i \in [h - M(k), h - M(k-1))$. The problem of determining the candidate literal edges is thus reduced to finding minimum values in a moving window. This can be done in amortized $\mathcal{O}(1)$ time and $\mathcal{O}(E(k))$ auxiliary space by keeping a list of ascending minima, along with their position in the sequence. Thus, by maintaining Q moving windows, *in parallel*, all the candidate literal edges can be generated in $\mathcal{O}(n\,Q)$ time and $\mathcal{O}(n)$ auxiliary space. We have thus proved the following lemma.

LEMMA 5.1. *BLPP extended with literal strings can be solved in $\mathcal{O}(n(s_{costs} + Q))$, under the assumption that literal lengths are encoded with an encoder with Q cost classes.*

Literal edges improve in a substantial way both compression ratio and decompression speed, as demonstrated in the following section.

5.2. The new bit-optimal LZ77-compressor. In this subsection we investigate several issues in the design of the best bit-optimal LZ77-compressor, thus improving over the FSG algorithm first proposed by Ferragina, Nitto, and Venturini [20], and consequently the bicriteria compressor described in section 4 that uses the FSG algorithm as a subroutine.

On integer encoders. We experimented with various integer encoders for the LZ77-phrases: Variable Byte (VByte), 4-Nibble (Nibble), and Elias' γ (Gamma) and δ (Delta) [40, 43]. We also introduced two variants of those encoders, called VByte-Fast and T-Nibble, which perform particularly well on LZ77-phrases.

The VByte-Fast encoder is a variant of Variable-Byte in which the maximum encodable integer is 2^{30} and in which the four indicator bits are expressed in binary in the two bits of the first byte.

TABLE 5.1
Space and time gains of bit-optimal strategy versus greedy strategy.

File	Encoder	Space gain	Time gain
Census	T-Nibble	13.7%	18.28%
	VByte-Fast	9.5%	13.48%
Dna	T-Nibble	18.0%	18.35%
	VByte-Fast	16.7%	12.00%
Mingw	T-Nibble	11.3%	16.72%
	VByte-Fast	8.1%	11.04%
Wikipedia	T-Nibble	15.5%	15.02%
	VByte-Fast	11.8%	8.81%

The T-Nibble encoder is a generalization of the Gamma encoder that makes the notion of cost classes explicit, and quite similar to cost classes of 4-Nibble when truncated to 1GiB (the maximum integer that may be encoded in our experiments), whence the "Truncated-Nibble" name. The idea underlying T-Nibble is to partition the range $[1, n]$ into subranges whose cardinalities are powers of 2. An integer i falling into the jth subrange $[s_j, s_j + 2^{k_j} - 1]$ is encoded as j in unary, followed by $i - s_j$ in binary using k_j bits. This scheme is quite flexible because subranges (i.e., k_j) may be expanded or contracted to adapt to the integer distribution at hand.

Encoders Nibble, T-Nibble, Gamma, and Delta are not *byte-aligned*, meaning that the start of an integer in a stream does not generally start at a byte's boundary, while VByte and VByte-Fast are byte-aligned. It is generally considerably faster to decode a sequence of byte-aligned integers because they do not require explicit shifting while being decoded. On the other hand, forgoing byte alignment usually implies higher compressions because they better adapt to the integer distribution at hand. In the following we thus refer to these byte-aligned encoders (i.e., VByte and VByte-Fast) as the *fast* ones, and the others as the *succinct* ones.

Bit-optimal versus greedy strategy. The impact of the bit-optimal strategy is well demonstrated by Table 5.1, in which we report the space savings obtained with respect to the widely popular greedy strategy whereby the rightmost longest copy is always selected. This minimizes the space occupancy of the greedily selected phrases.

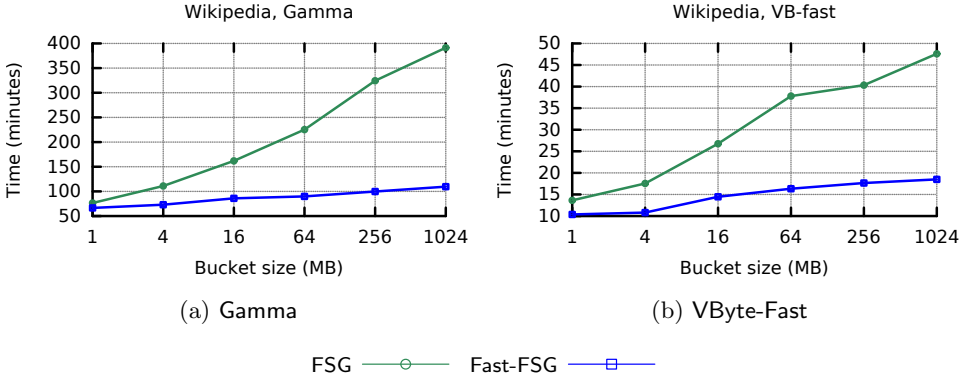
Bit-optimal is a clear winner over greedy on all of our datasets, producing parsings at least 10% smaller on average ($\approx 11.5\%$ with VByte-Fast, $\approx 14.6\%$ with T-Nibble) while being $\approx 14\%$ faster at decompression ($\approx 17\%$ with T-Nibble, $\approx 11.3\%$ with VByte-Fast). Working space was $\approx$ 59GiB of main memory. Using bit-optimal in lieu of greedy means that we can use a faster encoder without sacrificing compression ratios. Bit-optimal achieves this result by "adapting" parsing choices to the symbol ideal probability distribution of the underlying integer encoders. This phenomenon is clearly illustrated in Table 5.2, in which we report the sum of the empirical entropies of distances and lengths in the parsing, and the Kullback–Leibler (KL) divergence [32] with the encoder ideal probability distributions. In the bit-optimal parsing both the entropy and the KL-divergence are lower than with greedy. These results motivate our analysis of section 3.2 about ways to improve the computational efficiency of the bit-optimal strategy with respect to the previously proposed one [20], as discussed next.

Compression performances. We now experimentally compare the running time of the bit-optimal LZ77 algorithm when employing either the original FSG algo-

TABLE 5.2

Entropy and Kullback–Leibler (KL) divergence for the parsings computed with bit-optimal and greedy strategies.

Dataset	Bit-optimal		Greedy	
	Entropy	KL	Entropy	KL
Census (VByte-Fast)	23.89	8.53	26.83	11.37
Census (T-Nibble)	23.46	6.16	26.81	10.75
Dna (VByte-Fast)	24.65	6.69	27.33	10.89
Dna (T-Nibble)	24.23	4.19	27.23	9.22
Mingw (VByte-Fast)	23.99	6.54	26.22	7.91
Mingw (T-Nibble)	22.90	2.30	25.95	4.54
Wikipedia (VByte-Fast)	26.39	5.47	29.18	7.40
Wikipedia (T-Nibble)	25.90	2.70	29.12	5.60



(a) Gamma (b) VByte-Fast

FIG. 5.1. *Comparison between the novel Fast-FSG and FSG in parsing the dataset Wikipedia by using VByte-Fast and Gamma as integer encoders. Compression time is reported for varying bucket size.*

rithm for generating maximal edges [20] or our novel Fast-FSG algorithm, described in section 3.2. Figure 5.1 shows the results only for dataset Wikipedia, as the figures do not change significantly on the other datasets. We ran several experiments by increasing the bucket size of the bit-optimal compressor from 1MiB to 1GiB, in steps of powers of 4. We only report the results for the two integer encoders—namely, Gamma and VByte-Fast—that gave the lowest and highest performance gaps among all the encoders we tested.

In the plots, Fast-FSG exhibits a significantly lower time performance; the time-curve is approximately linear in the bucket size, experimentally confirming the theoretical constant amortized time per edge. On the contrary, FSG shows the superlinear behavior induced by its time cost of $\Theta(\log b)$ per edge, where b is the size of the bucket. In the Gamma case, the running time for the 1MiB bucket size is virtually the same for FSG and Fast-FSG, while their relative gap is already $\approx$ 4x for a 1GiB bucket size. In the VByte-Fast case, FSG is $\approx$ 1.3x to $\approx$ 2.5x slower than Fast-FSG.

The Fast-FSG approach is therefore instrumental in making the bit-optimal LZ77-compressor competitive in terms of compression time against the widely used and top performing compressors; indeed, our bit-optimal construction is on par with or faster than Lzma2, as we show in the last paragraph of this section.

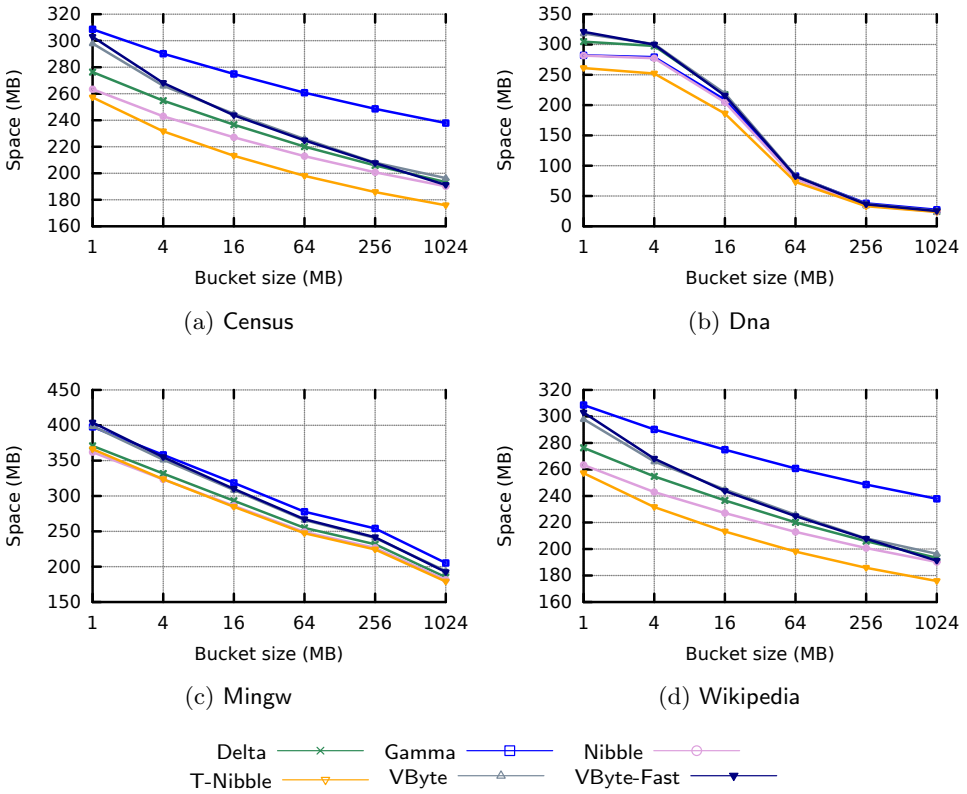


FIG. 5.2. *Compression ratios obtained by compressing every dataset with every integer encoder.*

Choosing the best integer encoders. For the sake of clarity we aim at establishing two integer-encoder candidates for the subsequent experiments, namely the one yielding the best compression ratio among the *fast* encoders and the one among the *succinct* encoders. They allow us first to establish the range of trade-offs achievable by our novel bit-optimal compressor and then to extend our achievements and analysis to our novel bicriteria compressor.

Results are summarized in Figure 5.2. The clear outliers are Delta and Gamma, which yield the worst compression ratio across all datasets. The most succinct encoder is T-Nibble almost everywhere, closely followed by Nibble. Both are byte-unaligned and thus "slow" encoders, although Nibble may carry a little advantage in decompression speed because it is "half-aligned" to 8-bit word boundaries. The byte-aligned ("fast") encoders are VByte and VByte-Fast, which are virtually indistinguishable compressionwise, so we pick VByte-Fast due to its potentially higher decoding speed.

Compression ratios improve almost logarithmically on the bucket size in three out of four datasets; the exception is Dna, which exhibits a sharp drop in compressed size when using buckets from 4 to 64 MB (2^{20} bytes). This is easily explained by the fact that Dna is a collection of genome families in which every family has genomes of different lengths but with very small differences in their contents; thus, the compressed size sharply drops when the bucket is large enough to back-reference a previous genome of the same family.

Overall, Figure 5.2 shows that compression ratios do improve with a longer bucket size, but the exact improvement depends on the peculiarities of the data being com-

pressed. This implies that trading decompression time versus compression ratio via the choice of the bucket size alone requires a deep understanding of the data being compressed.

Literal strings impact on compression ratio. We recall from section 5.1 that a literal string α of length ℓ is encoded by emitting plainly the string α, preceded by the integer ℓ encoded in binary within a fixed number of bits; in our tests we used either 8 or 16 bits, which limits the maximum length of a run to 2^8 and 2^{16}, respectively. Then the literal string is copied verbatim, followed by a 32-bit integer that encodes the number of copy-phrases to be processed until the next literal string occurs.

We compared the compressed size produced by the bit-optimal LZ77-compressor when using no literals, or allowing literal strings of length up to 2^8, or 2^{16}. This gives compressed sizes very close to each other on all the datasets except for Mingw, where the use of literals gives a sharp advantage. The use of 16-bit literals introduces the largest improvement, which is $\approx 16\%$ with VByte-Fast and $\approx 10\%$ with T-Nibble.

Moreover, literal strings generally improve decompression speed. The speed improvement is highly tied to compression ratio improvements; thus the decompression speed gain in Mingw is higher than those achieved with the other datasets. Those improvements in decompression time are quite substantial: for instance, with encoder VByte-Fast the ratio between the decompression time with no literals and with 16-bit literals ranges from ≈ 1.52 to ≈ 1.68, while for T-Nibble it ranges from ≈ 1.32 to ≈ 1.65. This behavior verifies our claims in section 5.1, where we argued that the use of literals should be more important on files showing incompressible portions, and it should be useful to speed up their decompression. In fact, the ratio between the number of characters outputted by literal-phrases (that is, the sum of all literal lengths) and the uncompressed size is close to zero for each dataset except Mingw, where it is $\approx 7\%$. Moreover, the mean literal length is larger in Mingw by a factor 10 (hundreds versus thousands), indicating that entire portions of Mingw are represented verbatim. Given these results, in the remaining experiments we fix our compressor to use 16-bit encodings of literals.

Decompression speed. In Figure 5.3 we show the decompression time taken by the two encoders T-Nibble and VByte-Fast selected on the basis of the results of the previous paragraphs. The behavior on Census and Wikipedia is the one that may intuitively be expected, with decompression time trending generally upwards with the bucket size due to the decrease of locality. Even there, however, the trend is not uniform, with a small dip at 4MiB: taking into account that the machine's L_2 cache has size 4MiB, this can be explained by the fact that the reduced decoding time (fewer phrases) is not counterbalanced in time by higher latencies as the cache is spacious enough to hold most back-references. Conversely, for Mingw the time is basically stable at around 1100 ms with VByte-Fast irrespective of the bucket size, while the trend is actually *improving* with T-Nibble, starting from around 2000 ms with 1MiB buckets but reducing to less than 1400 ms with the largest bucket size. The trend is even more striking for Dna, where it is apparent for both encoders: starting at around 2200 ms and 1300 ms with 1MiB buckets for T-Nibble and VByte-Fast, respectively, the decompression time decreases all the way to around 500 ms for 1GiB buckets. This is a very substantial decrease, by a factor above 4 and 2, respectively.

These results clearly show how the dependency between bucket size and decompression speed of the bit-optimal LZ77-compressor highly depends on the characteristics of the data being compressed. This clearly motivates our insistence on the bicriteria graph model that *separately*, and in light of these results correctly, takes

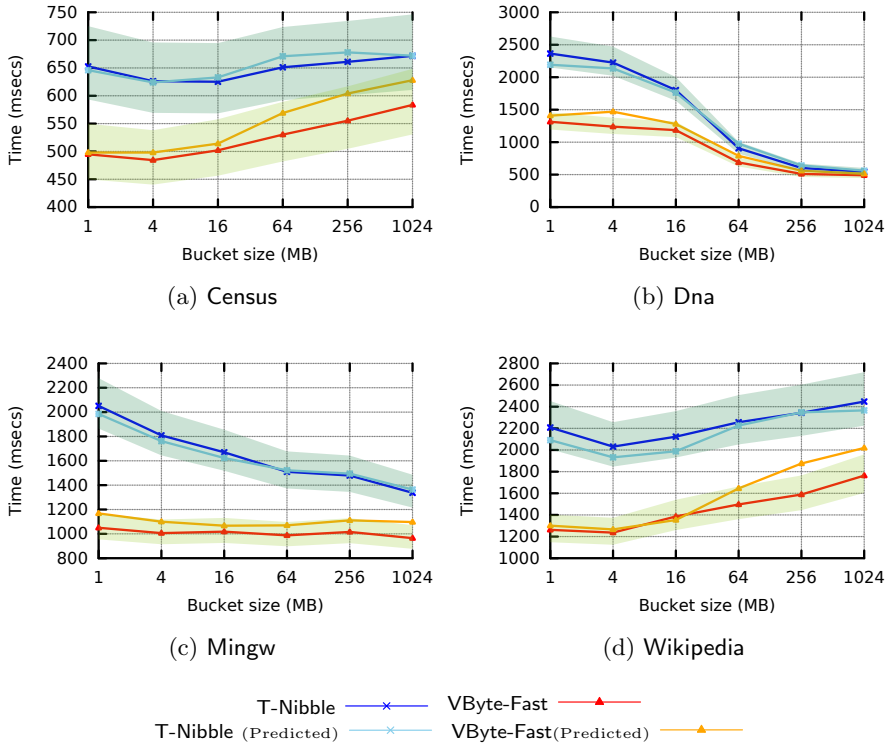


FIG. 5.3. *Decompression time by varying the bucket size. The plots also report the time predicted by our decompression-time model (see section* 5.3*) and a band around the decompression time capturing a relative error of* 10%.

into account for every LZ77-phrase both its space occupancy and its decoding time. In fact, enlarging the bucket size augments the graph with longer edges, thus resulting in higher time costs. Using these long edges in the parsing generally improves compression, but the effects on decompression time depend on the number of "shorter" edges they replace and on the fine details of the impacts on the cache misses in the memory hierarchy. Several different cases can present themselves in practice; for instance, the ratio between the highest and the lowest space cost of the edges in our graph $\mathcal{G}$ is on the order of 1–10, while the ratio between the highest and the lowest time cost of the edges is on the order of 100–300. The surprising behavior of Dna, and to a smaller degree of Mingw, is due to the fact that there are back-references that copy long portions of a genome, thus compensating the cache miss penalty induced by that copy. On the other hand, Wikipedia and Census are less repetitive; hence, back-references added by larger windows are not much longer, they save comparatively little space, and thus they do not compensate the cache miss penalties incurred by their decompression.

Our bicriteria graph model and bicriteria compressor take into account simultaneously and in a principled way all these issues; in fact they allow us to determine the required trade-off in space occupancy and decoding time independent of the fine characteristics of input data to be compressed.

Overall comparison. Table 5.3 reports the comparison between the two reference implementations of the bit-optimal LZ77-compressor (LzOpt hereafter), namely

TABLE 5.3

Comparison between bit-optimal compressor (LzOpt), bicriteria compressor (Bc-Zip), and state-of-the-art data compressors. For each dataset we highlight the parsing having the closest decompression time to Lz4.

Dataset	Compressor	Compressed size (MBytes)	Decompression time (msecs)
Census	LzOpt (T-Nibble)	38.08	776
	LzOpt (VByte-Fast)	40.19	572
	Bc-Zip (VByte-Fast, 556 ms)	40.38	549
	Bc-Zip (VByte-Fast, 454 ms)	44.42	462
	Bc-Zip (VByte-Fast, 370 ms)	110.43	365
	Gzip	48.23	2,472
	Lzma2	33.03	2,652
	Snappy	123.68	634
	Lz4	61.82	454
	Bzip2	39.96	15,054
	BigBzip	33.28	71,000
	Ppmd	38.70	38,000
Mingw	LzOpt (T-Nibble)	179.01	1,586
	LzOpt (VByte-Fast)	192.34	954
	Bc-Zip (VByte-Fast, 920 ms)	193.77	845
	Bc-Zip (VByte-Fast, 726 ms)	205.56	695
	Bc-Zip (VByte-Fast, 461 ms)	293.62	472
	Gzip	344.47	5,534
	Lzma2	187.68	8,323
	Snappy	461.00	891
	Lz4	384.67	726
	Bzip2	317.96	32,469
	BigBzip	222.22	152,000
	Ppmd	245.54	414,000
Dna	LzOpt (T-Nibble)	23.78	598
	LzOpt (VByte-Fast)	25.14	482
	Bc-Zip (VByte-Fast, 455 ms)	27.97	468
	Bc-Zip (VByte-Fast, 418 ms)	47.59	432
	Bc-Zip (VByte-Fast, 381 ms)	75.08	395
	Gzip	245.25	5,815
	Lzma2	17.62	1,681
	Snappy	448.67	1,301
	Lz4	333.74	1,007
	Bzip2	45.79	34,157
	BigBzip	42.02	152,000
	Ppmd	196.36	129,000
Wikipedia	LzOpt (T-Nibble)	175.86	3,080
	LzOpt (VByte-Fast)	191.19	1,748
	Bc-Zip (VByte-Fast, 1306 ms)	205.89	1460
	Bc-Zip (VByte-Fast, 973 ms)	270.35	1106
	Bc-Zip (VByte-Fast, 862 ms)	316.18	986
	Gzip	269.36	6,154
	Lzma2	166.16	9,871
	Snappy	422.80	1,093
	Lz4	309.51	862
	Bzip2	214.65	29,037
	BigBzip	150.88	151,000
	Ppmd	148.27	283,000

the ones based on the integer encoders T-Nibble and VByte-Fast, and the best-known compressors to date. Among the space-efficient compressors, Lzma2 is the best in terms of compressed size except on Wikipedia, where it is not too far from the best ones ($\approx$ 10%). Where Lzma2 is the champion, the gap with respect to LzOpt is, overall, acceptable, ranging from 25% on Dna, to 15% on Census, to 2% on Mingw; actually, LzOpt with T-Nibble is almost 5% more succinct than Lzma2 there. It should be remarked that Lzma2 uses a more sophisticated statistical encoder, whereas LzOpt is restricted to the use of stateless ones. On Wikipedia, LzOpt is about 15% worse than Lzma2 and 30% worse than the most succinct one which, however, has a decompression speed two orders of magnitude larger. In all cases, LzOpt exhibits much better decompression times than Lzma2; even using the slower encoder T-Nibble, the ratio is around 3x for Census, Dna, and Wikipedia, and $>$ 5x for Mingw.

Compared with the fastest compressors (Snappy and Lz4), parsings obtained with LzOpt are much more succinct: the relative gap in compressed space ranges from $\approx$ 60% (Census) to over 1300% (Dna). Decompression speed is competitive, especially if the slightly less succinct VByte-Fast encoder is taken into account: decompression time is better on Dna, very close in Census and Mingw (relative gaps of $\approx$ 26% and $\approx$ 32%, respectively), and slower but on the same order of magnitude in Wikipedia (gap of about 100%).

In section 5.4 we close this speed gap between the bicriteria-based scheme and Snappy or Lz4. In order to do this, however, we need to develop reasonable estimates of the time costs of each arc, which is discussed next.

5.3. Decompression-time model. In sections 2.2 and 5.1 we estimated the time needed to process a phrase p as

$$t(p) = \begin{cases} t(d,\ell) = t_{\mathcal{D}}(d,\ell) + n_M(\ell)\, t(d) + \ell\, t_{\mathcal{C}}, & p = \langle d, \ell \rangle, \\ t_B + t_U(\ell) + \ell\, t_{\mathcal{C}}, & p = \langle \alpha, \ell \rangle_L. \end{cases}$$

In order to effectively deploy this model in an actual implementation, every term appearing in the equation must be precisely instantiated with respect to the machine executing the decompression and the software routines used for copying bytes. We now thus define more precisely the term $n_M(\ell)$ that estimates the number of cache line borders that are crossed while copying a phrase of length ℓ. More precisely, the term $n_M(\ell)$ takes (probabilistically) into account whether a copy of length ℓ does not cross a cache border (so $n_M(\ell) = 1$) or does (and so $n_M(\ell) = 2$). If $\ell \geq L_i$, then the cache border is always crossed and thus two cache misses are always paid. If instead $\ell < L_i$, the cache line border may or may not be crossed, depending on ℓ and on the position of the first character in the line. Since the position is essentially unpredictable, we assume it as randomly drawn from $[1, L_i]$ with uniform distribution, which yields an average number of cache misses of $1 + \min(1, (\ell - 1)/L_i)$. In addition, provided that our copy-routines access the memory in blocks of 8 consecutive bytes, we estimate the number of cache misses triggered by a copy of length ℓ as

$$n_M(\ell) = 1 + \min\left(1, \left\lceil \frac{\ell - 1}{8} \right\rceil \frac{8}{L_i}\right).$$

Other hardware-dependent terms, such as $t(d)$, are evaluated in an automated fashion by means of a *calibration tool* that derives the values of these parameters through a series of microbenchmarks, in which each penalty term (memory latency $t(d)$, phrase/literal copy-time $t_{\mathcal{C}}$, literal-phrase unpack t_U, integer decode time $t(d, \ell)$,

branch misprediction t_B) is individually measured. More precisely, we get the memory hierarchy configuration (number of cache levels and their sizes) and the corresponding memory latency by deploying the tool `lmbench`.[5] Since the time penalties cannot be individually timed due to their extremely short duration, we evaluated them *in bundle*, timing a considerable number of them, and taking the average per event.

Predicting execution times of processes is a notoriously hard task, and achieving predictions within a 10% average relative error is usually considered satisfactory [24]. In Figure 5.3, the time predicted by our decompression-time model is plotted side-by-side with the actual decompression time; for convenience, a band around the predicted decompression time is also shown capturing a relative error of 10%. As the figure shows, most of the predicted decompression times fall well inside the band. Overall, the average relative error is $\approx 5.6\%$ (VByte-Fast $\approx 8.3\%$, T-Nibble $\approx 3\%$), the maximum error prediction is $\approx 18.5\%$, and the vast majority of predictions (over 85%) stays below 10%.

5.4. Plugging everything into Bc-Zip. Our implementation of the Bc-Zip compressor largely follows the scheme presented in section 4. However, in order to attain better compression times and space requirements, we have added to the software a number of implementation tricks that we now succinctly describe.

Algorithmic improvements. The Lagrangian dual DWCSPP (cf. section 4.2) is solved through the use of Kelley's cutting-plane algorithm, which in turn requires the solution of $\mathcal{O}(\log n)$ instances of the BLPP. A naive implementation of this strategy requires generating $\mathcal{G}$'s edges multiple times (one per instance) via the Fast-FSG algorithm, thus significantly increasing the overall compression time. We have therefore implemented a caching strategy that lowers those edge-generations to 2 and still takes $\mathcal{O}(n)$ space.

The idea here is to avoid storing the distances in LZ77-phrases that are not actually needed for shortest path computations; rather, we keep, for each edge, its length and an ID encoding the two cost classes of its time/space costs. We can show that this takes 2 bits on average per edge, and thus $\mathcal{O}(n \log n)$ bits (hence $\mathcal{O}(n)$ memory words) to store the pruned $\widetilde{\mathcal{G}}$. Pick any length cost class and denote by $e(v)$ the endpoint of the edge leaving from v with that cost; if there is no such edge, set $e(v) = v$. Clearly $e(v)$ is nondecreasing in v because if there is an edge (v, v'), then the edge $(v+1, v')$ exists, too, thanks to Property 3.2. So $e(v+1) \geq e(v)$ and hence the length of $(v+1, v')$ can be gap-encoded as $e(v+1) - e(v)$. Since there are at most n edges of a given cost class, $e(1) = 1$ and $e(n) = n$, it follows that all those gaps sum up to n, thus taking at most $2n$ bits using the Gamma code. Since the cost classes are $\mathcal{O}(\log n)$ and cost-IDs can be encoded implicitly, the space bound holds.

Since distances in LZ77-phrases are not needed for shortest path computation, we need to execute Fast-FSG at most twice: the first time to generate the pruned graph, and the second time when the computed parsing has to be written on disk, in order to derive the phrase-distances of its edges.

In this section we illustrate Bc-Zip's performance in compression time and approximation factor. Our implementation slightly differs from the description given in section 4, and thus it turns out to be slightly worse in the achieved approximation guarantees. This stems from the fact that here we halt the Lagrangian dual computation when the relative gap between the upper bound (the value $\varphi = \max_\lambda \varphi_B(\lambda) = \varphi_B(\lambda^+)$) and the lower bound (the value $\varphi' = L(\pi^+, \lambda^+)$, where π^+ is a λ^+-optimal

[5] http://www.bitmover.com/lmbench/

solution) is less than $\epsilon = 10^{-6}$. The following lemma shows that, in this case, the final solution is guaranteed to have space bounded by $(1 + 2\,\epsilon)z^\star + s_{\max}$.

LEMMA 5.2. *If the basis given as input to the path-swap procedure is an* $(1+\epsilon)$*-approximation of the Lagrangian dual, then the resulting solution* π *is such that* $s(\pi) \leq (1 + 2\epsilon)\, z^\star + s_{\max}$ *and* $t(\pi) \leq \mathcal{T} + 2\, t_{\max}$.

Proof. Let λ^+ be the λ value maximizing φ_B at the beginning of the last iteration, and let φ and φ' be defined as above. Let also $z^\star \geq \varphi'$ be the cost of the optimal solution. By assumption, $\varphi < (1+\epsilon)\varphi'$. Let (π_A, π_B) be the path-swapped solutions for some swapping-point v. By following the same argument as in Lemma 4.8, it holds that $L(\lambda^+, \pi_A) + L(\lambda^+, \pi_B) \leq 2\,\varphi + s_{\max} + \lambda^+ t_{\max}$ and $L(\lambda^+, \pi_A),\ L(\lambda^+, \pi_B) \geq \varphi'$. Thus, it follows that any path-swapped solution cannot have λ^+-cost greater than

$$2\varphi - \varphi' + s_{\max} + \lambda^+ t_{\max} = (1 + 2\epsilon)\varphi' + s_{\max} + \lambda^+ t_{\max}.$$

Due to Lemma 4.9, there is a swapped path π such that $t(\pi) \in [\mathcal{T} + t_{\max},\ \mathcal{T} + 2t_{\max}]$. Since $L(\pi, \lambda^+) = s(\pi) + \lambda^+(t(\pi) - \mathcal{T}) \leq (1 + 2\epsilon)\varphi' + s_{\max} + \lambda^+ t_{\max}$, and since $t(\pi) \geq \mathcal{T} + t_{\max}$, it follows that $s(\pi) \leq (1 + 2\epsilon)\varphi' + s_{\max} \leq (1 + 2\epsilon)z^\star + s_{\max}$. ∎

So let us consider the gap between the solution space cost $s(\pi)$ and the lower bound given by φ' on the set of compressions carried out for the overall comparison reported in the next paragraph. In particular, let us consider the relative gap $\frac{s(\pi)-\varphi'}{\varphi'}$: the maximum such value among all compressions is $\approx 1.09 \cdot 10^{-6}$, which is extremely tight as it amounts to an absolute difference of just 270 bytes out of hundreds of megabytes. In many cases, the compressor exploits the opportunity of (very slightly) violating the bound to produce solutions better than the lower bound. We also point out that the maximum gap is just $\approx 65\%$ of the maximum gap predicted by Lemma 5.2, and ≈ 30x the maximum gap predicted by Theorem 4.1. The average/maximum/minimum compression time is about 147/173/100 minutes. We recall that the compression time is given by the time taken for two full runs of the Fast-FSG algorithm (about 30 minutes each), the path-swap (about 10 seconds), and $t + 1$ bit-optimal computations with the graph-caching technique, where t is the number of iterations of the Lagrangian dual (about 8 minutes each). The average/maximum/minimum number of iterations was 8.84/12/3. Since the number of iterations decreases with increasing ϵ, this parameter can be tuned to offer a compression-time/compression-ratio trade-off, as suggested by Lemma 5.2. According to our estimates, compressing the dataset with $\epsilon = 10^{-3}$ instead of 10^{-6} would result in solutions with an average/maximum/minimum absolute space gap with respect to solutions computed with $\epsilon = 10^{-6}$ of just 18/88/0.4 kilobytes, with approximately 30% lower running times. We point out that this low sensitivity on ϵ is a byproduct of the path-swap procedure. In fact, if the feasible solution of the dual basis were chosen instead of their path-swap, such average/maximum/minimum gaps would be 56/69/12 *MB*, three orders of magnitude higher.

Efficiency and effectiveness of the algorithm.

Overall comparison. In order to explore the whole range of trade-offs offered by Bc-Zip, in our tests we compressed each dataset several times, for both VByte-Fast and T-Nibble, with time bounds ranging from the decompression time of the time-optimal parsing to the decompression time of the space-optimal one; the resulting compression levels are plotted in Figure 5.4. The first remarkable result is just how wide the range actually is: on Mingw, for instance, it spans from ≈ 300 ms to ≈ 1400 ms timewise,

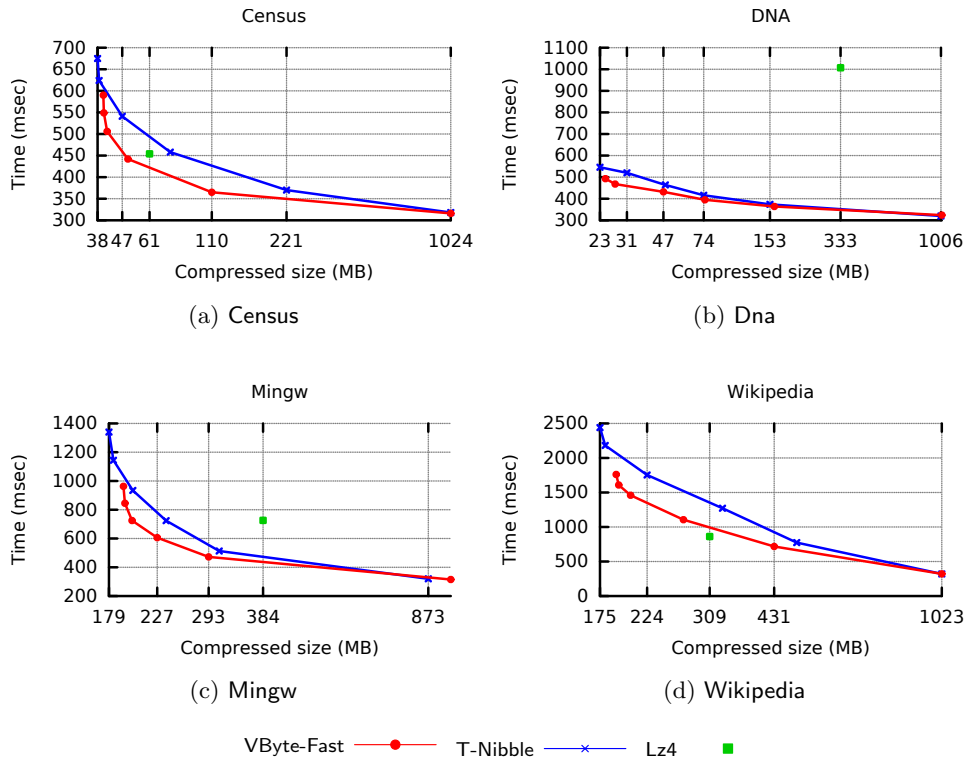


FIG. 5.4. *Time/space trade-off curve obtained with Bc-Zip, by varying the decompression time bound, and Lz4.*

and from ≈ 976 MB to ≈ 179 MB spacewise. Another interesting result is that, at least for our testing machines, T-Nibble is competitive against VByte-Fast only when maximum compression is required; otherwise, the latter delivers more succinct parsings for the same decompression time. This is due to T-Nibble's relatively slow decoding, which forces Bc-Zip to replace LZ77 copies for literals in order to meet the decompression-time budget; conversely, VByte-Fast's fast decoder does not have much impact on decompression time, and thus Bc-Zip is left with a larger set of options to choose from. A final, crucial observation is that the shape of the curve is, in general, far from linear, especially on the leftmost part (with varying degree: more accentuated with Mingw, less with Wikipedia). That is, varying the decompression time has little impact on compressed size when more succinct parsings are considered, while it may have more impact when less space-efficient parsings are taken into account. This shows that the two questions at the beginning of the paper about the sensitivity of compressed file size to decompression speed were well-posed and meaningful in practice: the time/space trade-off is not linear, and a small change in one resource may induce a significant change in the other.

Finally, in order to directly compare Bc-Zip against the state-of-the-art compressors adopted in storage systems such as Hadoop and Bigtable, we compressed each dataset by setting its decompression-time bound as the decompression time of the parsings generated by Lz4 (highlighted entries in Table 5.3). The average time model accuracy is ≈ 4.5% (VByte-Fast ≈ 5.4%, T-Nibble ≈ 3.7%). The results are reported

in Table 5.3 and show that Bc-Zip is extremely competitive with Lz4: with the same (to within a few percentage points) or less decompression time, it improves the compressed file size by $\approx 28\%$ in Census, by $\approx 47\%$ in Mingw, and by $\approx 91\%$ in Dna, where it is also more than twice as fast. Bc-Zip is only bested in Wikipedia by a meager $\approx 2\%$ in file size, and by $\approx 12\%$ in speed. Overall, Bc-Zip performs more consistently than well-engineered, and widely used, compressors such as Lz4. This is because these have been designed over a set of trade-offs fixed once and for all and independent of the text being compressed. Thanks to its principled design, Bc-Zip is more capable of adapting to different texts; furthermore, it offers a handy "knob" that the user can turn to explore the time/space trade-off, possibly setting it to different sweet spots for different applications without the need to re-engineer a different compressor for each. We therefore believe that this is a nice success case of a win-win situation between algorithmic theory and engineering.

6. Conclusion. Our bicriteria compressor is able to compute efficiently any fixed trade-off in space occupancy and decoding time by taking into account the time and space characteristics of all Lempel–Ziv parsings of the input text to be compressed.

We remark that our algorithmic techniques can be easily extended to variants of the LZ77-compressor that deploy parsers that refer to a *bounded* compression window. Experimenting with the effect of window size is left as an interesting future work.

An interesting theoretical open question involves extending our results to *statistical* encoders like Huffman or arithmetic coders to encode phrase distances and lengths. They do not necessarily satisfy the nondecreasing cost property, since long distances and lengths may occur more frequently than shorter ones. We argue that it is not trivial to design bit-optimal and bicriteria compressors for these encoding functions because their codeword lengths change as it changes the set of distances and lengths used in the parsing process.

Appendix. Omitted proofs.

THEOREM 4.3. *For any fixed $\epsilon > 0$ there exists a multiplicative $\left(\epsilon, \frac{\epsilon}{2}\right)$-approximation scheme for the BDCP that takes $\mathcal{O}\left(\frac{1}{\epsilon}\left(n \log^2 n + \frac{1}{\epsilon^2}\log^4 n\right)\right)$ time and $\mathcal{O}\left(n + \frac{1}{\epsilon^3}\log^4 n\right)$ space complexity.*

Proof. The main idea behind this theorem is to solve WCSPP with the additive approximation algorithm of Theorem 4.1, thus obtaining a path $\pi^\star$. If $s_{\max}/\epsilon \le \varphi^\star$ and $t_{\max}/\epsilon \le \mathcal{T}$ (recall that $\varphi^\star$ is computed by the algorithm), then $\pi^\star$ is a solution with the required accuracy, i.e., $s(\pi^\star) \le \varphi^\star + s_{\max} \le (1+\epsilon)\varphi^\star$ and $t(\pi^\star) \le \mathcal{T} + 2t_{\max} \le (1+2\epsilon)\mathcal{T}$, and we can return it. Otherwise, we execute an exhaustive search algorithm that takes subquadratic time because it explores a very small space of candidate solutions.

In fact, we have that either $s_{\max}/\epsilon > \varphi^\star$ or $t_{\max}/\epsilon > \mathcal{T}$. Assuming that the first case holds (the other case is symmetric and leads to the same conclusion), we can find the optimal path by enumerating a small set of paths in $\mathcal{G}$ via a breadth-first visit delimited by a pruning condition.

The key idea is to prune a subpath π', going from node 1 to some node v', if $s(\pi') > s_{\max}/\epsilon$; this should be a relatively easy condition to satisfy, since $s_{\max}$ is the maximum space cost of one single edge (hence of an LZ77-phrase). If this condition holds, then $s(\pi') > s_{\max}/\epsilon > \varphi^\star$ (see before); hence π' cannot be optimal and thus can be discarded. We can also prune π' upon finding a path π'' that arrives at a farther node $v'' > v'$ while requiring the same compressed space and decompression time of π' (Lemma 3.5).

Hence, all paths π that are not pruned have $s(\pi) \leq s_{\max}/\epsilon$ and thus $t(\pi) \leq s(\pi)t_{\max} \leq s_{\max}t_{\max}/\epsilon$ (just observe that every edge has integral time cost in the range $[1, t_{\max}]$). We can therefore adopt a dynamic programming approach, à la knapsack [37], which computes the exact optimal solution (not just an approximation) by filling a bidimensional matrix m of size $S \times U$, where $S = s_{\max}/\epsilon$ is the maximum feasible space cost and $U = S\,t_{\max}$ is the maximum time cost admitted for a candidate solution/path. Entry $m[s,t]$ stores the farthest node in $\mathcal{G}$ reachable from 1 by a path π with $s = s(\pi)$ and $t = t(\pi)$. These entries are filled in L rounds, where $L \leq s_{\max}/\epsilon$ is the maximum length of the optimal path (just observe that every edge has integral space cost in the range $[1, s_{\max}]$). Each round ℓ constructs the set X_ℓ of paths having length ℓ and starting from node 1 that are candidates to be the optimal path. These paths are generated by extending the paths in $X_{\ell-1}$, via the visit of the forward star of their last nodes, in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space according to the Fast-FSG algorithm (see section 3). Each generated path π' is pruned if either $s(\pi') > s_{\max}/\epsilon$ or its last node is to the left of node $m[s(\pi'), t(\pi')]$; otherwise we set that node into that entry (Lemma 3.5). The algorithm goes to the next round by setting X_ℓ as the paths remaining after the pruning.

As far as the time complexity of this process is concerned, we note that $|X_\ell| \leq S\,U = \mathcal{O}\big(\log^3 n/\epsilon^2\big)$. The forward star of each node needed for X_ℓ can be generated by creating the pruned $\widetilde{\mathcal{G}}$ in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space. Since we have a total of $L \leq s_{\max}/\epsilon = \mathcal{O}(\log n/\epsilon)$ rounds, the total time is $\mathcal{O}\big(\log^4 n/\epsilon^3 + n \log^2 n/\epsilon\big)$.

As a final remark, note that one could instead generate the forward stars of all nodes in $\widetilde{\mathcal{G}}$, which requires $\mathcal{O}(n \log n)$ time *and* space, and then use them as needed. This would simplify and speed up the algorithm, achieving $\mathcal{O}(n \log n/\epsilon)$ total time; however, this would increase the working space to $\mathcal{O}(n \log n)$, and a superlinear requirement in the size n of the input string $\mathcal{S}$ to be compressed is best avoided in the context in which these algorithms are more likely to be used. ∎

REFERENCES

[1] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir, *A model for hierarchical memory*, in Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC), 1987, pp. 305–314, https://doi.org/10.1145/28395.28428.

[2] A. Aggarwal, A. K. Chandra, and M. Snir, *Hierarchical memory with block transfer*, in Proceedings of the 28th Annual Symposium on Foundations of Computer Science (FOCS), 1987, pp. 204–216, https://doi.org/10.1109/SFCS.1987.31.

[3] A. Aggarwal, B. Schieber, and T. Tokuyama, *Finding a minimum-weight k-link path in graphs with the concave Monge property and applications*, Discrete Comput. Geom., 12 (1994), pp. 263–280, https://doi.org/10.1007/BF02574380.

[4] A. Aggarwal and J. S. Vitter, *The input/output complexity of sorting and related problems*, Comm. ACM, 31 (1988), pp. 1116–1127, https://doi.org/10.1145/48529.48535.

[5] B. Alpern, L. Carter, E. Feig, and T. Selker, *The uniform memory hierarchy model of computation*, Algorithmica, 12 (1994), pp. 72–109, https://doi.org/10.1007/BF01185206.

[6] M. A. Bender and M. Farach-Colton, *The LCA problem revisited*, in Proceedings of the 4th Latin American Symposium on Theoretical Informatics Symposium (LATIN), 2000, pp. 88–94, https://doi.org/10.1007/10719839_9.

[7] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. S. Aiyer, *Apache Hadoop goes realtime at Facebook*, in Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, 2011, pp. 1071–1080, https://doi.org/10.1145/1989323.1989438.

[8] N. Brisaboa, A. Fariña, G. Navarro, and M. Esteller, *(s,c)-dense coding: An optimized compression code for natural language text databases*, in String Processing and Information Retrieval, SPIRE 2003, Lecture Notes in Comput. Sci. 2857, Springer, Berlin, Heidelberg,

2003, pp. 122–136, https://doi.org/10.1007/978-3-540-39984-1_10.

[9] G. S. Brodal, R. Fagerberg, M. Greve, and A. López-Ortiz, *Online sorted range reporting*, in Proceedings of the 20th International Symposium on Algorithms and Computation (ISAAC), 2009, pp. 173–182, https://doi.org/10.1007/978-3-642-10631-6_19.

[10] M. Burrows and D. J. Wheeler, *A Block-sorting Lossless Data Compression Algorithm*, SRC Research Report 124, Digital, Palo Alto, CA, 1994.

[11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, *Bigtable: A distributed storage system for structured data*, ACM Trans. Comput. Syst., 26 (2008), 4, https://doi.org/10.1145/1365815.1365816.

[12] Z. Cohen, Y. Matias, S. Muthukrishnan, S. C. Sahinalp, and J. Ziv, *On the temporal HZY compression scheme*, in Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2000, pp. 185–186.

[13] G. Cormode and S. Muthukrishnan, *Substring compression problems*, in Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2005, pp. 321–330, https://dl.acm.org/citation.cfm?id=1070432.1070478.

[14] U. Drepper, *What Every Programmer Should Know about Memory*, 2007, http://www.akkadia.org/drepper/cpumemory.pdf.

[15] I. Dumitrescu and N. Boland, *Improved preprocessing, labeling and scaling algorithms for the weight-constrained shortest path problem*, Networks, 42 (2003), pp. 135–153, https://doi.org/10.1002/net.10090.

[16] P. Elias, *Universal codeword sets and representations of the integers*, IEEE Trans. Inform. Theory, 21 (1975), pp. 194–203, https://doi.org/10.1109/TIT.1975.1055349.

[17] M. Farach and M. Thorup, *String matching in Lempel-Ziv compressed strings*, Algorithmica, 20 (1998), pp. 388–404, https://doi.org/10.1007/PL00009202.

[18] P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino, *Boosting textual compression in optimal linear time*, J. ACM, 52 (2005), pp. 688–713, https://doi.org/10.1145/1082036.1082043.

[19] P. Ferragina, I. Nitto, and R. Venturini, *On optimally partitioning a text to improve its compression*, Algorithmica, 61 (2011), pp. 51–74, https://doi.org/10.1007/s00453-010-9437-6.

[20] P. Ferragina, I. Nitto, and R. Venturini, *On the bit-complexity of Lempel–Ziv compression*, SIAM J. Comput., 42 (2013), pp. 1521–1541, https://doi.org/10.1137/120869511.

[21] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979.

[22] S. Golomb, *Run-length encodings*, IEEE Trans. Inform. Theory, 12 (1966), pp. 399–401, https://doi.org/10.1109/TIT.1966.1053907.

[23] G. Y. Handler and I. Zang, *A dual algorithm for the constrained shortest path problem*, Networks, 10 (1980), pp. 293–309, https://doi.org/10.1002/net.3230100403.

[24] L. Huang, J. Jia, B. Yu, B.-G. Chun, P. Maniatis, and M. Naik, *Predicting execution time of computer programs using sparse polynomial regression*, in Proceedings of the 24th International Conference on Neural Information Processing Systems (NIPS'10), 2010, pp. 883–891.

[25] J. Kärkkäinen, P. Sanders, and S. Burkhardt, *Linear work suffix array construction*, J. ACM, 53 (2006), pp. 918–936, https://doi.org/10.1145/1217856.1217858.

[26] J. Katajainen and T. Raita, *An analysis of the longest match and the greedy heuristics in text encoding*, J. ACM, 39 (1992), pp. 281–294, https://doi.org/10.1145/128749.128751.

[27] O. Keller, T. Kopelowitz, S. L. Feibish, and M. Lewenstein, *Generalized substring compression*, Theoret. Comput. Sci., 525 (2014), pp. 42–54, https://doi.org/10.1016/j.tcs.2013.10.010.

[28] J. E. Kelley, Jr., *The cutting-plane method for solving convex programs*, J. Soc. Indust. Appl. Math., 8 (1960), pp. 703–712, https://doi.org/10.1137/0108053.

[29] D. Kempa and S. J. Puglisi, *Lempel–Ziv factorization: Simple, fast, practical*, in Proceedings of the Fifteenth Workshop on Algorithm Engineering and Experiments (ALENEX), SIAM, Philadelphia, 2013, pp. 103–112, https://doi.org/10.1137/1.9781611972931.9.

[30] S. R. Kosaraju and G. Manzini, *Compression of low entropy strings with Lempel–Ziv algorithms*, SIAM J. Comput., 29 (1999), pp. 893–911, https://doi.org/10.1137/S0097539797331105.

[31] S. Kreft and G. Navarro, *On compressing and indexing repetitive sequences*, Theoret. Comput. Sci., 483 (2013), pp. 115–133, https://doi.org/10.1016/j.tcs.2012.02.006.

[32] S. Kullback and R. A. Leibler, *On information and sufficiency*, Ann. Math. Statistics, 22 (1951), pp. 79–86, https://doi.org/10.1214/aoms/1177729694.

[33] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Dover Books on Mathematics Series, Dover Publications, Mineola, NY, 2001.

[34] E. L. Lloyd and S. S. Ravi, *Topology control problems for wireless ad hoc networks*, in Handbook of Approximation Algorithms and Metaheuristics, Chapman & Hall/CRC Comput. Inf. Sci. Ser., Chapman & Hall/CRC, Boca Raton, FL, 2007, pp. 67-1–67-20, https://doi.org/10.1201/9781420010749.

[35] F. Luccio and L. Pagli, *A model of sequential computation with pipelined access to memory*, Math. Systems Theory, 26 (1993), pp. 343–356, https://doi.org/10.1007/BF01189854.

[36] M. V. Marathe, R. Ravi, R. Sundaram, S. S. Ravi, D. J. Rosenkrantz, and H. B. Hunt, III, *Bicriteria network design problems*, J. Algorithms, 28 (1998), pp. 142–171, https://doi.org/10.1006/jagm.1998.0930.

[37] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*, John Wiley & Sons, New York, 1990.

[38] K. Mehlhorn and M. Ziegelmann, *Resource constrained shortest paths*, in Proceedings of the 8th Annual European Symposium on Algorithms (ESA), 2000, pp. 326–337, https://doi.org/10.1007/3-540-45253-2_30.

[39] J. A. Nelder and R. Mead, *A simplex method for function minimization*, Comput. J., 7 (1965), pp. 308–313, https://doi.org/10.1093/comjnl/7.4.308.

[40] D. Salomon, *Data Compression: The Complete Reference*, 4th ed., Springer-Verlag, 2006, https://doi.org/10.1007/978-1-84628-603-2.

[41] E. J. Schuegraf and H. S. Heaps, *A comparison of algorithms for data base compression by use of fragments as language elements*, Inform. Storage Ret., 10 (1974), pp. 309–319, https://doi.org/10.1016/0020-0271(74)90069-2.

[42] J. S. Vitter and E. A. M. Shriver, *Algorithms for parallel memory,* II: *Hierarchical multilevel memories*, Algorithmica, 12 (1994), pp. 148–169, https://doi.org/10.1007/BF01185208.

[43] I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*, 2nd ed., Morgan Kaufmann, San Francisco, CA, 1999.

[44] J. Ziv and A. Lempel, *A universal algorithm for sequential data compression*, IEEE Trans. Inform. Theory, 23 (1977), pp. 337–343, https://doi.org/10.1109/TIT.1977.1055714.

[45] J. Ziv and A. Lempel, *Compression of individual sequences via variable-rate coding*, IEEE Trans. Inform. Theory, 24 (1978), pp. 530–536, https://doi.org/10.1109/TIT.1978.1055934.