# On the bit-complexity of Lempel-Ziv compression*

Paolo Ferragina Igor Nitto Rossano Venturini

Dipartimento di Informatica, Università di Pisa, Italy

**Abstract**

One of the most famous and investigated lossless data-compression schemes is the one introduced by Lempel and Ziv about 30 years ago [37]. This compression scheme is known as "dictionary-based compressor" and consists of squeezing an input string by replacing some of its substrings with (shorter) codewords which are actually pointers to a dictionary of phrases built as the string is processed. Surprisingly enough, although many fundamental results are nowadays known about the speed and effectiveness of this compression process (see e.g. [23, 29] and references therein), *"we are not aware of any parsing scheme that achieves optimality when the* `LZ77`*-dictionary is in use under any constraint on the codewords other than being of equal length"* [29, pag. 159]. Here optimality means to achieve the *minimum* number of bits in compressing *each individual* input string, *without* any assumption on its generating source. In this paper we investigate three issues pertaining to the bit-complexity of LZ-based compressors, and we design algorithms which achieve bit-optimality in the compressed output size by taking efficient/optimal time and optimal space. These theoretical results will be sustained by some experiments that will compare our novel LZ-based compressors against the most popular compression tools (like `gzip`, `bzip2`) and state-of-the-art compressors (like the *booster* of [13, 12]).

*It has been partially supported by Yahoo! Research, Italian MIUR Italy-Israel FIRB Project and PRIN MAINSTREAM. The authors' address is Dipartimento di Informatica, L.go B. Pontecorvo 3, 56127 Pisa, Italy. email: {ferragina,nitto,rossano}@di.unipi.it

## 1 Introduction

The problem of *lossless* data compression consists of compactly representing data in a format that can be faithfully recovered from the compressed file. Lossless compression is achieved by taking advantage of the *redundancy* present often in the data generated by either humans or machines. One of the most famous lossless data-compression schemes is the one introduced by Lempel and Ziv in the late 70s, and indeed many (non-)commercial programs are currently based on it— like `gzip`, `zip`, `pkzip`, `arj`, `rar`, just to cite a few. This compression scheme is known as *dictionary-based compressor*, and consists of squeezing an input string $S[1,n]$ by replacing some of its substrings with (shorter) *codewords* which are actually pointers to a dictionary of phrases. The dictionary can be either *static* (in that it has been constructed before the compression starts) or *dynamic* (in that it is built as the input string is compressed). The well-known `LZ77` and `LZ78` compressors, proposed by Lempel and Ziv in [37, 38], and all their variants [30], are interesting examples of *dynamic* dictionary-based compressors. In `LZ77`, and its variants, the dictionary consists of all substrings starting in the last $M$ scanned positions of the text, where $M$ is called the *window size* and possibly depends on the text length. Each codeword consists of a triple $\langle d, \ell, c\rangle$ where $d$ is the relative *offset* of the copied phrase ($d \leq M$), $\ell$ is its length and $c$ is the single (new) character following it. In `LZ78`, the dictionary is built upon phrases extracted from the previously scanned prefix of the input string, and each codeword consists of a pair $\langle \texttt{id}, c\rangle$ where `id` is the *identifier* of the dictionary phrase and $c$ is the character following that phrase in the string.

Many theoretical and experimental results have been dedicated to LZ-compressors in these thirty years (see e.g. [30] and references therein); and, although today there are alternative solutions to the lossless data-compression problem (e.g., Burrows-Wheeler compression and Prediction by Partial Matching [36]), dictionary-based compression is still widely used for its unique combination of compression power and compression/decompression speed. Over the years dictionary-based compression has also gained importance as a general algorithmic tool, being employed in the design of compressed text indexes [27], in *universal* clustering [5] or classification tools [39], in designing *optimal* pre-fetching mechanisms [34], and in streaming or on-the-fly compression applications [8, 17].

In this paper we address some key issues which arise when dealing with the output-size in bits of LZ-parsing schemes. The classical `LZ77` and `LZ78` algorithms adopt a *greedy parsing* of the input string: namely, at each step, they take the *longest* dictionary phrase which is a prefix of the currently unparsed string suffix. This greedy parsing can be computed in $O(n \log \sigma)$ time and $O(M)$ space [15].[1] The greedy parsing is also optimal with respect to the *number of phrases* in which $S$ can be parsed by any suffix-complete dictionary (like `LZ77`) or, a small variation of it (called *flexible*-parsing [25]), is optimal for prefix-complete dictionaries (like `LZ78`). Of course, the number of parsed phrases influences the compression ratio and, indeed, various authors [37, 23] proved that greedy parsing achieves asymptotically the *(empirical) entropy* of the source generating the input string $S$. However, these fundamental results have *not yet closed* the problem of optimally compressing $S$ because the optimality in the number of parsed phrases *is not necessarily equal* to the optimality in the number of bits output by the final compressor on *each individual* input string $S$. In fact, if the phrases are compressed via an *equal-length* encoder, like in [23, 30, 37], then the produced output is *bit optimal*. But if one aims for higher compression, *variable-length encoders* should be taken into account (see e.g. [36, 11], and the software `gzip` [19]), and in this situation the greedy-parsing scheme is *no longer optimal* in terms of the number of bits output by the final compressor. Starting from these premises we address in this paper four main problems, both on the theoretical and the experimental side, which pertain with the bit-optimal compression of the input string $S$ via parsers

[1] Recently, [9] showed how to achieve the optimal $O(n)$ time and space when the alphabet has size $O(n)$ and the window is unbounded, i.e. $M = n$.

that deploy the LZ77-dictionary and an unbounded window within which the copied phrases can be selected (namely it is $M = n$). These parsers will be hereafter called "LZ77-parsers". Our results extend to the LZ78-dictionary and to some other LZ-variants too (see Section 6).

**Problem 1.** Let us consider the greedy LZ77-parser, and assume that we encode every parsed phrase $w_i$ with a variable-length encoder. The value of $\ell_i = |w_i|$ is in some sense fixed by the greedy choice, being the length of the longest phrase occurring in the current LZ77-dictionary. Conversely, the value of $d_i$ depends on the position of the copy of $w_i$ in $S$. In order to minimize the bits output by the final compressor, the parser should obviously select the *closest* copy of each phrase $w_i$ in $S$, and thus the smallest possible $d_i$. Surprisingly enough, known parsers are time optimal but not bit-optimal, because they select an arbitrary or the leftmost occurrence of the longest copied phrase (see [9] and references therein), or they select the closest copy but take $O(n \log n)$ suboptimal time [1, 24]. In Section 3 we provide an elegant, yet simple, algorithm which computes at each parsing step the closest copy of the longest dictionary phrase in $O(n \frac{\log \sigma}{\log \log n})$ overall time and $O(n)$ space (Section 3, Lemma 1). This is optimal in terms of time/space performance when the alphabet has size *polylog(n)* (i.e. almost all texts of practical interest), and it is also optimal in terms of the compressed output size produced by any LZ77-parsing scheme based on the greedy selection rule.

**Problem 2.** How good is the greedy LZ77-parsing of $S$ whenever the compression cost is measured in terms of bits produced in output? Not surprisingly, we show that the greedy selection of the longest dictionary phrase at each parsing step is not optimal! But surprisingly, we show that the loss in using the greedy parser with respect to the parser that achieves bit-optimality in the compressed output size is *not* negligible, and diverges by a multiplicative factor $\Omega(\log n / \log \log n)$, which is unbounded asymptotically (Section 4). Also, we show that this lower-bound is tight up to a factor $\Theta(\log \log n)$, and support these theoretical figures with some experimental results, see Table 1 below, which stress the practical importance of finding the bit-optimal parsing of $S$.

**Problem 3.** How much efficiently (in time and space) can we compute the bit-optimal LZ77-parsing of $S$? Several solutions are indeed known for this problem but they are either inefficient [31, 16], in that they take $\Theta(n^2)$ worst-case time and space, or they are approximate [20], or they rely on heuristics [22, 32, 4, 6, 16] which do not provide any guarantee on the time/space performance of the compression process. This is the reason why Rajpoot and Sahinalp stated in [29, pag. 159] that *"We are not aware of any on-line or off-line parsing scheme that achieves optimality when the* LZ77*-dictionary is in use under any constraint on the codewords other than being of equal length"*. In this paper we investigate this question by considering a general class of variable-length codeword encodings which are typically used in data compression (e.g. `gzip`) and in the design of search engines and compressed indexes [27, 30, 36]. Our final result is a time efficient and space optimal solution for the problem above (Theorem 8). Due to space limitations, we will detail our results only for the LZ77-parser (dictionary), and discuss other parsing schemes (like LZ78) in the last Section 6.

Technically speaking, we follow [31] and model the search for a bit-optimal LZ77-parsing of an input string $S$ as a *single-source shortest path* problem (shortly, SSSP) on a *weighted* DAG $\mathcal{G}(S)$ consisting of $n$ nodes, one per character of $S$, and $e$ edges, one per possible LZ77-parsing step. Every edge is weighted according to the length in bits of the codeword adopted to compress the corresponding phrase. Since these codewords are tuples of integers (see above), we consider a natural class of codeword encoders which satisfy the so called *increasing cost property*: the larger is the integer to be encoded, the longer is the codeword. This class encompasses most of the encoders used in the literature to design data compressors (see [11] and `gzip` [19]), compressed full-text indexes [27] and search engines [36]. We prove new combinatorial properties for this SSSP-problem and show that the computation of the SSSP in $\mathcal{G}(S)$ can be restricted onto a subgraph $\widetilde{\mathcal{G}}(S)$ whose structure depends on the integer-encoding functions adopted to compress the LZ77-phrases, and whose size is *provably smaller* than the complete graph generated by [31] (see Theorem 7). Additionally, we design an algorithm that solves the SSSP on the subgraph $\widetilde{\mathcal{G}}(S)$ without materializing it *all at once*, but it creates and explores its edges *on-the-fly* in optimal $O(1)$ amortized time per edge and $O(n)$ optimal space overall. As a result, our novel LZ77-compressor achieves bit-optimality in $O(n)$ optimal working space and in time proportional to $|\widetilde{\mathcal{G}}(S)|$ (hence, it is optimal in its size). The latter is $O(n \log n)$ for a large class of integer encoders, like Elias and Fibonacci codes [36, 11], and it is the optimal $O(n)$ for (most of) the encodings used by gzip [19]. This is the first result providing a *positive answer* to Rajpoot-Sahinalp's question above!

**Problem 4.** How much efficient are *in practice* our novel LZ77-parsers compared to known compressors? To establish this, we have taken several freely available text collections, and compared our LZ77-based compressors against the classic gzip and bzip2, as well as against the state-of-the-art *boosting* compressor of [13, 12]. Table 1 reports some experimental figures. Let us first consider algorithm Fixed-LZ77, which uses an unbounded window and equal-length encoders for the distance of the copied phrases: its compression performance shows that an unbounded window may introduce a significant compression gain wrt to a bounded one, as used by gzip and bzip2 (see e.g. HTML), thus witnessing the presence in current (Web/text) collections of surprisingly many long repetitions at large distances. Then we consider Rightmost-LZ77 (Problem 1), and notice that it improves Fixed-LZ77: as expected, variable-length encoders achieve higher compression ratios. This was the starting point of our theoretical investigation! Finally, we tested BitOptimal-LZ77 (Problem 3) finding that it improves Rightmost-LZ77, as theoretically predicted in Problem 2. Surprisingly, BitOptimal-LZ77 significantly improves bzip2 (which uses a bounded window) and comes close to the booster (which uses an unbounded window). Additionally, since BitOptimal-LZ77 adopts the same decompression algorithm of gzip, it retains its *fast decompression speed* which is at least one order of magnitude faster than bzip2 and the booster (see table below). This is a nice combination which makes BitOptimal-LZ77 practically relevant for a wide range of applications in which the paradigm is "compress once & decompress many times" (like in Web search engines and IR systems), or where the decompression system is less powerful than the compressor one (like a server that distributes data to clients, possibly mobile phones). These results *resort* LZ-based approaches to compress large textual collections, and thus pave the way to future algorithmic engineering research devoted to design proper variable-length encoders for BitOptimal-LZ77, which possibly depend onto the structural features of the input data to be compressed (see Section 6).

| Compressor | english [14] | HTML [3] | C/C++/Java src [14] | Avg Decompr. time (sec) |
|---|---:|---:|---:|---:|
| gzip -9 | 37.52% | 20.09% | 23.29% | 0.7 |
| bzip2 -9 | 28.40% | 10.63% | 19.78% | 6.3 |
| boosterOpt | 20.62% | 3.89% | 17.36% | 20.2 |
| Fixed-LZ77 | 26.19% | 4.98% | 24.63% | 0.8 |
| Rightmost-LZ77 | 23.81% | 4.27% | 20.14% | 0.9 |
| BitOptimal-LZ77 | 21.62% | 3.87% | 17.62% | 0.9 |

Table 1: Each text collection consists of 50 Mbytes of data. All the experiments were executed on a 2.6 GHz Pentium 4, with 1.5 GB of main memory, and running Fedora Linux.

## 2 Notation and terminology

Let $S[1, n]$ be a string drawn from an alphabet $\Sigma$ of size $\sigma$. We will use $S[i]$ to denote the $i$th symbol of $S$; $S[i : j]$ to denote the substring (also called the *phrase*) extending from the $i$th to the $j$th symbol in $S$ (extremes included); and $S_i = S[i : n]$ to denote the $i$-th suffix of $S$.

In the rest of the paper we will concentrate on LZ77-compression; our results are easily generalizable to arbitrary window sizes (possibly constant, like in gzip), and other LZ-variants (see Section 6). Dictionary-based compression works in two intermingled phases: *parsing* and *encoding.*

Let $w_1, w_2, \ldots, w_{i-1}$ be the phrases in which a prefix of $S$ has been already parsed. At this step, the LZ77-dictionary consists of all substrings of $S$ starting in the last $M$ positions of $w_1 w_2 \cdots w_{i-1}$, where $M$ is called the *window size*. The LZ77-parser selects the next phrase according to the so called *longest match heuristic*: that is, this phrase is taken as the *longest* phrase in the current dictionary which prefixes the remaining suffix of $S$. This will be hereafter called *greedy parsing*. After such a phrase is selected, the parser adds one further symbol to it and thus forms the next phrase $w_i$ of $S$'s parsing. In the rest of the paper, and for simplicity of exposition, we will restrict to the case of *unbounded window* (i.e. $M = n$), and to the LZ77-variant which avoids the additional symbol per phrase. This means that $w_i$ is represented by the integer pair $\langle d_i, \ell_i \rangle$, where $d_i$ is the relative *offset* of the copied phrase $w_i$ within the prefix $w_1 \cdots w_{i-1}$ and $\ell_i$ is its length $|w_i|$. Every first occurrence of a new symbol $c$ is encoded as $\langle 0, c \rangle$.

Once phrases are identified and represented via pairs of integers, their components are compressed via *variable-length integer encoders* which eventually produce the compressed output of $S$ as a sequence of bits. In order to study and design bit-optimal parsing schemes, we therefore need to deal with such integer encoders. Let $f$ be an integer-encoding function that maps any integer $x \in [n]$ into a (bit-)codeword $f(x)$ whose length is denoted by $|f(x)|$ bits. In this paper we consider variable-length encodings which use longer codewords for larger integers:

**Property 1 (Increasing Cost Property)** *For any $x, y \in [n]$ it is $x \leq y$ iff $|f(x)| \leq |f(y)|$.*

This property is satisfied by most of known integer encoders— like equal-length codewords, Elias codes [36], Fibonacci's codes [11]— which are used to design data compressors [30], compressed full-text indexes [27] and search engines [36].

## 3 An efficient and bit-optimal greedy LZ77-parsing

Let $f$ and $g$ be two integer-encoders which satisfy the Increasing Cost Property (possibly $f = g$). We denote by $\mathtt{LZ}_{f,g}(S)$ the compressed output produced by the greedy-parsing strategy in which we have used $f$ to compress the distance $d_i$, and $g$ to compress the length $\ell_i$ of any parsed phrase $w_i$. Thus, $\mathtt{LZ}_{f,g}(S)$ encodes any phrase $w_i$ in $|f(d_i)| + |g(\ell_i)|$ bits. Given that the parsing is the greedy one, $\ell_i$ is in some sense fixed (to be the length of the longest copy), so we minimize the overall bit complexity of $\mathtt{LZ}_{f,g}$ by minimizing the *distance* $d_i$ of $w_i$'s copy in $S$. If $p_i$ is the starting position of $w_i$ in $S$ (namely $S[p_i, p_i + \ell_i - 1] = w_i$), many copies of the phrase $w_i$ could be present in $S[1, p_i - 1]$. To minimize $|\mathtt{LZ}_{f,g}|$ we should choose the copy which is the closest one to $p_i$, and thus requires the minimum number of bits to encode its distance $d_i$ (recall the assumption $M = n$).

In this section we propose an elegant, yet simple, algorithm that selects the rightmost copy of each phrase $w_i$ in $O(n \log \sigma / \log \log n)$ time. This algorithm is the fastest known in the literature [9, 24], and results optimal for alphabets with a *polylog*$(n)$ size (i.e., almost all texts in practice). It requires the suffix tree $\mathcal{ST}$ of $S$, preprocessed to support constant-time lca-queries, and the LZ77-parsing of $S$ which consists of, say, $k \leq n$ phrases. We say that a node $u$ of $\mathcal{ST}$ is *marked* iff the string spelled out by the root-to-$u$ path in $\mathcal{ST}$ is equal to some phrase $w_i$. In this case we use the notation $u_{p_i}$ to denote the node marked by phrase $w_i$ which starts at position $p_i$ of $S$. Since the same node may be marked by different phrases, but any phrase marks just one node, the total number of marked nodes is bounded by the number of phrases, hence $k$. Furthermore, if a node is assigned with many phrases, since the greedy LZ77-parsing takes the longest one, it must be the case that every such occurrences of $w_i$ is followed by a distinct character. So the number of phrases assigned to the same marked node is bounded by $\sigma$.

All marked nodes can be computed in linear time in $O(k)$ time by executing $k$ lca-queries on $\mathcal{ST}$. Let us now define $\mathcal{ST}_\mathcal{C}$ as the contracted version of $\mathcal{ST}$, namely a tree whose internal nodes are the marked nodes of $\mathcal{ST}$ and whose leaves are the leaves of $\mathcal{ST}$. The parent of any node in $\mathcal{ST}_\mathcal{C}$ is its lowest marked ancestor in $\mathcal{ST}$. It is simple to build $\mathcal{ST}_{\mathcal{C}}$ in linear time via a top-down visit of $\mathcal{ST}$. $\mathcal{ST}_{\mathcal{C}}$ consists of $O(k)$ internal nodes and $n$ leaves.

Given the properties of suffix trees, we can now rephrase our problem as follows: for each position $p_i$, we need to compute the largest position $x_i$ which is smaller than $p_i$ and whose leaf in $\mathcal{ST}_{\mathcal{C}}$ lies within the subtree rooted at $u_{p_i}$. Our algorithm processes the input string $S$ from left to right and, at each position $j$, it maintains the following invariant: the parent $v$ of any leaf in $\mathcal{ST}_{\mathcal{C}}$ stores the maximum position $h < j$ such that the leaf labeled $h$ is attached to $v$. Maintaining this invariant is trivial: after that position $j$ is processed, $j$ is assigned to the leaf parent of the leaf labeled $j$ in $\mathcal{ST}_{\mathcal{C}}$. The key point is how to compute the position $x_i$ of the rightmost-copy of $w_i$ whenever we discover that $j$ is the starting position of a phrase (i.e. $j = p_i$ for some $i$). In this case, the algorithm visits the subtree of $\mathcal{ST}_{\mathcal{C}}$ rooted at $u_j$ and computes the maximum position stored in its marked nodes. By the invariant, this position is the rightmost copy of the phrase $w_i$. This process takes $O(n + \sigma \sum_{i=1}^{k} \#(u_{p_i}))$ time, where $\#(u_{p_i})$ is the number of marked nodes in the subtree rooted at $u_{p_i}$ in $\mathcal{ST}_{\mathcal{C}}$. In fact, by construction, there can be at most $\sigma$ repetitions of the same phrase in the `LZ77`-parsing of $S$, and for each of them the algorithm performs a visit of the corresponding subtree.

As a final step we prove that $\sum_{i=1}^{k} \#(u_{p_i}) = O(n)$. By construction of suffix trees, the depth of $u_{p_i}$ is smaller than $\ell_i = |w_i|$, and each (marked) node of $\mathcal{ST}_{\mathcal{C}}$ is visited as many times as the number of its (marked) ancestors in $\mathcal{ST}_{\mathcal{C}}$ (with their multiplicities). For each (marked) node $u_{p_i}$, this number can be bounded by $\ell_i = O(\ell_i)$. Summing up on all nodes, we get $\sum_{i=1}^{k} O(l_i) = O(n)$. Thus, the above algorithm requires $O(\sigma \times n)$ time, which is trivially optimal whenever $\sigma = O(1)$. Actually, we are able to further reduce the time complexity to $O(n \log \sigma / \log \log n)$ by properly combining a slightly modified variant of the tree covering procedure of [18] with a dynamic Range Maximum Query data structure [26, 35] applied on properly composed arrays of integers. Notice that this improvement leads our algorithm to require the optimal $O(n)$ time, for alphabets whose size is poly-logarithmic in $n$. Due to the lack of space the description of this solution is deferred to Appendix A.

**Lemma 1** *Given a string $S[1, n]$ drawn from an alphabet of size $\sigma$, we can design an algorithm that computes the greedy* `LZ77`*-parsing of $S$ and reports the rightmost copy of each phrase in $O(n \frac{\log \sigma}{\log \log n})$ time and $O(n)$ space.*

## 4 On the bit-efficiency of the greedy `LZ77`-parsing

We have already noticed that $\mathtt{LZ}_{f,g}(S)$ is not necessarily bit optimal, so we will hereafter use $\mathtt{OPT}_{f,g}(S)$ to denote the $(f, g)$-*optimal parser*, namely the one that parses $S$ into a sequence of phrases which are drawn from the `LZ77`-dictionary and which *minimize* the total number of bits produced by the encoders $f$ and $g$. Of course $|\mathtt{LZ}_{f,g}(S)| \geq |\mathtt{OPT}_{f,g}(S)|$, but this does not provide us with any estimate of how much worse the greedy parsing can be with respect to the bit-optimal one. In what follows we identify an infinite family of strings $S$ for which $\frac{|\mathtt{LZ}_{f,g}(S)|}{|\mathtt{OPT}_{f,g}(S)|} = \Omega(\frac{\log n}{\log \log n})$, so the gap may be asymptotically unbounded thus stressing the need for an $(f, g)$-optimal parser, as requested by [29].

Our argument holds for any choice of $f$ and $g$ from the family of encoding functions that represent an integer $x$ with a bit string of size $\Theta(\log x)$ bits (thus the well-known Elias' and Fibonacci's coders belong to this family). Taking inspiration from the proof of Lemma 4.2 in [23], we consider the infinite family of strings $S_l = ba^l\ c^{2^l}\ ba\ ba^2\ ba^3 \ldots ba^l$, parameterized in the positive value $l$. The greedy `LZ77`-parser partitions $S_l$ as[2]: $(b)$ $(a)$ $(a^{l-1})$ $(c)$ $(c^{2^l-1})$ $(ba)$ $(ba^2)$ $(ba^3)$ $\ldots$ $(ba^l)$, where the symbols forming a parsed phrase have been delimited within a pair of brackets. Thus it copies the latest $l$ phrases from the beginning of $S_l$ and takes at least $l\,|f(2^l)| = \Theta(l^2)$ bits.

[2]Recall the variant of `LZ77` we are considering in this paper, which uses just a pair of integers per phrase, and thus drops the char following that phrase in $S$. See section 2.

A parsimonious parser, called rOPT, selects the copy of $ba^{i-1}$ (with $i > 1$) from its immediately previous occurrence thus parsing $S_l$ as: $(b)$ $(a)$ $(a^{l-1})$ $(c)$ $(c^{2^l-1})$ $(b)$ $(a)$ $(ba)$ $(a)$ $(ba^2)$ $(a)$ $\ldots$ $(ba^{l-1})$ $(a)$. Hence $\texttt{rOPT}(S_l)$ takes $|g(2^l)| + |g(l)| + \sum_{i=2}^{l}[|f(i)| + |g(i)| + f(0)] + O(l) = O(l \log l)$ bits.

**Lemma 2** *There exists an infinite family of strings such that, for any of its elements $S$, it is* $|\texttt{LZ}_{f,g}(S)| \geq \Theta(\log|S|/\log\log|S|)\ |\texttt{OPT}_{f,g}(S)|$.

**Proof:** Since $\texttt{OPT}(S_l) \leq \texttt{rOPT}(S_l)$, we can conclude that: $\frac{|\texttt{LZ}_{f,g}(S_l)|}{|\texttt{OPT}_{f,g}(S_l)|} \geq \frac{|\texttt{LZ}_{f,g}(S_l)|}{|\texttt{rOPT}(S_l)|} \geq \Theta\left(\frac{l}{\log l}\right)$. Since $|S_l| = 2^l + l^2 - O(l)$, we have that $l = \Theta(\log|S_l|)$ for sufficiently long strings. ■

The experimental results reported in Table 1 show that this gap is not negligible in practice too, just look at the entries Fixed-LZ77 and BitOptimal-LZ77. Additionally we can prove that this lower bound is tight up to a $\log\log|S|$ multiplicative factor, by easily extending to Property 1 and to the LZ77-dictionary (which is dynamic), a result proved in [21] for static dictionaries. Precisely (details in the full paper), we can show that $\frac{|\texttt{LZ}_{f,g}(S)|}{|\texttt{OPT}_{f,g}(S)|} \leq \frac{|f(n)|+|g(n)|}{|f(0)|+|g(0)|}$, which is upper bounded by $O(\log n)$ because $|S| = n$, $|f(n)| = |g(n)| = \Theta(\log n)$ and $|f(0)| = |g(0)| = O(1)$.

## 5 Bit-Optimal Parsing and SSSP-problem

Following [31], we model the design of a bit-optimal LZ77-parsing strategy for a string $S$ as a Single-Source Shortest Path problem (shortly, SSSP-problem) on a weighted DAG $\mathcal{G}(S)$ defined as follows. Graph $\mathcal{G}(S) = (V, E)$ has one vertex per symbol of $S$ plus a dummy vertex $v_{n+1}$, and its edge set $E$ is defined so that $(v_i, v_j) \in E$ iff (1) $j = i + 1$ or (2) the substring $S[i : j - 1]$ occurs in $S$ starting from a (previous) position $p < i$. Clearly $i < j$ and thus $\mathcal{G}(S)$ is a DAG. Every edge $(v_i, v_j)$ is labeled with the pair $\langle d_{i,j}, \ell_{i,j}\rangle$ which is set to $\langle 0, S[i]\rangle$ in case (1), or it is set to $\langle p - i, j - i\rangle$ in case (2). The second case corresponds to copying a phrase longer than one single character.

It is easy to see that the edges outgoing from $v_i$ denote all possible parsing steps that can be taken by any parsing strategy which uses a LZ77-dictionary. Hence, there exists a one-to-one correspondence between paths from $v_1$ to $v_{n+1}$ in $\mathcal{G}(S)$ and LZ77-parsings of the whole string $S$. If we weight every edge $(v_i, v_j) \in E$ with an integer $c(v_i, v_j) = |f(d_{i,j})| + |g(\ell_{i,j})|$, which accounts for the cost of encoding its label (phrase) via the encoding functions $f$ and $g$, then the length in bits of the encoded parsing is equal to the cost of the corresponding weighted path in $\mathcal{G}(S)$. The problem of determining $\texttt{OPT}_{f,g}(S)$ is thus reduced to computing the shortest path from $v_1$ to $v_{n+1}$ in $\mathcal{G}(S)$.

Given that $\mathcal{G}(S)$ is a DAG, its shortest path from $v_1$ to $v_{n+1}$ can be computed in $O(|E|)$ time and space. However, this is $\Theta(n^2)$ in the worst case (take e.g. $S = a^n$ [31, 20]) thus resulting inefficient and actually un-usable in practice even for strings of few Megabytes. In what follows we show that the computation of the SSSP can be restricted to a subgraph of $\mathcal{G}(S)$ whose size depends on the choice of $f$ and $g$ satisfying Property 1, and is $O(n \log n)$ for most known integer-encoding functions. Then we will design efficient algorithms and data structures that will allow us to generate this subgraph *on-the-fly* by taking $O(1)$ amortized time per edge and $O(n)$ space overall. These algorithms will be therefore time-and-space optimal for the subgraph in hand, and will provide the first positive answer to Rajpoot-Sahinalp's question we mentioned at the beginning of this paper (see [29, pag. 159]).

### 5.1 A useful, small, subgraph of $\mathcal{G}(S)$

We use $FS(v)$ to denote the *forward star* of a vertex $v$, namely the set of vertices pointed to by $v$ in $\mathcal{G}(S)$; and we use $BS(v)$ to denote the *backward star* of $v$, namely the set of vertices pointing to $v$ in $\mathcal{G}(S)$. Since $\mathcal{G}(S)$ is a DAG, all of its edges $(v_i, v_j)$ are oriented rightward and thus $i < j$. Actually the indices of the vertices in $FS(v)$ and $BS(v)$ form a *contiguous* range:

**Fact 3** *Given a vertex $v_i$, it is $FS(v_i) = \{v_{i+1} \ldots, v_{i+x-1}, v_{i+x}\}$ and $BS(v_i) = \{v_{i-y} \ldots, v_{i-2}, v_{i-1}\}$. Furthermore, $x, y$ are smaller than the length of the longest repeated substring in $S$.*

**Proof:** By definition of $(v_i, v_{i+x})$, string $S[i : i+x-1]$ occurs at some position $p < i$ in $S$. Any prefix $S[i : k-1]$ of $S[i : i+x-1]$ also occurs at that position $p$, thus $v_k \in FS(v_i)$. The bound on $x$ immediately derives from the definition of $(v_i, v_{i+x})$. A similar argument holds for $BS(v_i)$. ■

This means that if an edge does exist in $\mathcal{G}(S)$, then exist also all edges which are *nested* within it and are incident into one of its extremes. The following property relates the indices of the vertices $v_j \in FS(v_i)$ with the cost of their connecting edge $(v_i, v_j)$, and not surprisingly shows that the smaller is $j$ (i.e. shorter edge), the smaller is the cost of encoding the phrase $S[i : j-1]$ (see Appendix B).[3]

**Fact 4** *Given a vertex $v_i$, for any pair of vertices $v_{j'}, v_{j''} \in FS(v_i)$ such that $j' < j''$, we have that $c(v_i, v_{j'}) \leq c(v_i, v_{j''})$. The same property holds for $v_{j'}, v_{j''} \in BS(v_i)$.*

Given these monotonicity properties, we are ready to characterize a special subset of the vertices in $FS(v_i)$, and their connecting edges.

**Definition 5** *An edge $(v_i, v_j) \in E$ is called*

**(1)** $d-$maximal *iff the next edge from $v_i$ takes more bits to encode its distance: $|f(d_{i,j})| < |f(d_{i,j+1})|$.*

**(2)** $\ell-$maximal *iff the next edge from $v_i$ takes more bits to encode its length: $|g(l_{i,j})| < |g(l_{i,j+1})|$.*

*Edge $(v_i, v_j)$ is called* maximal *if it is either d-maximal or $\ell$-maximal: thus $c(v_i, v_j) < c(v_i, v_{j+1})$.*

The number of maximal edges depends on the functions $f$ and $g$ (which satisfy Property 1). Let $Q(f, n)$ (resp. $Q(g, n)$) be the number of different codeword lengths generated by $f$ (resp. $g$) when applied to integers in the range $[n]$. We can partition $[n]$ into contiguous sub-ranges $I_1, I_2, \ldots, I_{Q(f,n)}$ such that the integers in $I_i$ are mapped by $f$ to codewords *(strictly) shorter than* the codewords for the integers in $I_{i+1}$. Similarly, $g$ partitions the range $[n]$ in $Q(g, n)$ contiguous sub-ranges.

**Lemma 6** *There are at most $Q(f, n) + Q(g, n)$ maximal edges outgoing from any vertex $v_i$.*

**Proof:** By Fact 3, vertices in $FS(v_i)$ have indices in a range $R$, and by Fact 4, $c(v_i, v_j)$ is monotonically non-decreasing as $j$ increases in $R$. Moreover we know that $f$ (resp. $g$) cannot change more than $Q(f, n)$ (resp. $Q(g, n)$) times, so that the statement follows. ■

To speed up the computation of a `SSSP` from $v_1$ to $v_{n+1}$, we construct a subgraph $\widetilde{\mathcal{G}}(S)$ of $\mathcal{G}(S)$ which is formed by maximal edges only, it is smaller than $\mathcal{G}(S)$ and contains one of those `SSSP`.

**Theorem 7** *There exists a shortest path in $\mathcal{G}(S)$ from $v_1$ to $v_{n+1}$ that traverses maximal edges only.*

**Proof:** By contradiction assume that every such shortest path contains at least one non-maximal edge. Let $\pi = v_{i_1} v_{i_2} \ldots v_{i_k}$, with $i_1 = 1$ and $i_k = n+1$, be one of these shortest paths, and let $\gamma = v_{i_1} \ldots v_{i_r}$ be the longest initial subpath of $\pi$ which traverses maximal edges only. Assume w.l.o.g. that $\pi$ is the shortest path maximizing the value of $|\gamma|$. We know that $(v_{i_r}, v_{i_{r+1}})$ is a non-maximal edge, and thus we can take the maximal edge $(v_{i_r}, v_j)$ that has the same cost. By definition of maximal edge, it is $j > i_{r+1}$; furthermore, we must have $j < n+1$ because we assumed that no path is formed by maximal edges only. Now, since $\mathcal{G}(S)$ is a DAG and indices in $\pi$ are increasing, it must exist an index $i_h \geq i_r$ such that the index of that maximal edge $j$ lies in $[i_h, i_{h+1}]$. Since $(v_{i_h}, v_{i_{h+1}})$ is an edge of $\pi$, it does exist the edge $(v_j, v_{i_{h+1}})$ (by Fact 3), and by Fact 4 on $BS(v_{i_{h+1}})$ we can conclude that $c(v_j, v_{i_{h+1}}) \leq c(v_{i_h}, v_{i_{h+1}})$. Consequently, the path $v_{i_1} \cdots v_{i_r} v_j v_{i_{h+1}} \cdots v_{i_k}$ is also a shortest path but its longest initial subpath of maximal edges consists of $|\gamma| + 1$ vertices, which is a contradiction! ■

[3]Recall that $c(v_i, v_j) = |f(d_{i,j})| + |g(\ell_{i,j})|$, if the edge does exist, otherwise we set $c(v_i, v_j) = +\infty$.

Theorem 7 implies that the distance between $v_1$ and $v_{n+1}$ is the same in $\mathcal{G}(S)$ and $\widetilde{\mathcal{G}}(S)$, with the advantage that computing SSSP in $\widetilde{\mathcal{G}}(S)$ can be done faster and in reduced space, because $|FS(v)| \leq Q(f,n) + Q(g,n)$ (Lemma 6). Thus, subgraph $\widetilde{\mathcal{G}}(S)$ consists of $n+1$ vertices and at most $n(Q(f,n) + Q(g,n))$ edges. For Elias' codes [10], Fibonacci's codes [11], and most practical integer encoders used for search engines and data compressors [30, 36], it is $Q(f,n) = Q(g,n) = O(\log n)$. Therefore $|\widetilde{\mathcal{G}}(S)| = O(n \log n)$, so it is smaller than the complete graph built and used by previous papers [31, 20, 16]. For the encoders used in `gzip`, it is $Q(f,n) = Q(g,n) = O(1)$ and $|\widetilde{\mathcal{G}}(S)| = O(n)$.

## 5.2 An efficient bit-optimal parser

From a high level, our solution is a variant of a classic linear-time algorithm for SSSP over a DAG (see [7, Section 24.2]), here applied to work on the subgraph $\widetilde{\mathcal{G}}(S)$. Therefore its correctness follows directly from Theorem 24.5 of [7] and our Theorem 7. However, the key difficulty in implementing this approach consists of how to *generate on-the-fly and efficiently (in time and space) the maximal edges outgoing from vertex* $v_i$. We will refer to this problem as the *forward-star generation* problem, and use FSG for brevity. In what follows we consider the case $\sigma \leq n$, and show that FSG takes $O(1)$ amortized time per edge and $O(n)$ space in total. In case of a larger alphabet, we need to add $T_{sort}(n,\sigma)$ time because of the sorting/remapping of $S$'s symbols into $[n]$. Since we have $n$ vertices, with no more than $Q(f,n) + Q(g,n)$ maximal edges each (Lemma 6), we will obtain the following:

**Theorem 8** *Given a string $S[1,n]$ drawn from an alphabet of size $\sigma$, and two integer-encoding functions $f$ and $g$ that satisfy Property 1, there exists a compressor that computes the $(f,g)$-optimal* LZ77*-parsing of $S$ in $O(n(Q(f,n) + Q(g,n)) + T_{sort}(n,\sigma))$ time and $O(n)$ space in the worst case.*

We know that the edges outgoing from $v_i$ can be partitioned into no more than $Q(f,n)$ groups according to the distance from $S[i]$ of the copied string they represent (proof of Lemma 6). Let $I_1, I_2, \ldots, I_{Q(f,n)}$ be the intervals of distances such that all distances in $I_k$ are encoded with the same number of bits by $f$. Take now the $d$-maximal edge $(v_i, v_{h_k})$ for the interval $I_k$. We can infer that substring $S[i : h_k - 1]$ is the *longest* substring having a copy at distance within $I_k$ because, by Definition 5 and Fact 4, any edge following $(v_i, v_{h_k})$ denotes a longer substring which must lie in a subsequent interval (by $d$-maximality of $(v_i, v_{h_k})$), and thus must have longer distance from $S[i]$. Once $d$-maximal edges are known, the computation of the $\ell$-maximal edges is then easy because it suffices to further decompose the edges between successive $d$-maximal edges, say between $(v_i, v_{h_{k-1}+1})$ and $(v_i, v_{h_k})$, according to the distinct values assumed by the encoding function $g$ on the lengths in the range $[h_{k-1}, \ldots, h_k - 1]$. This takes $O(1)$ time per $\ell$-maximal edge, because it needs some algebraic calculations, and the corresponding copied substring can then be inferred as a prefix of $S[i : h_k - 1]$ (details in the full paper).

So, let us concentrate on the computation of $d$-maximal edges outgoing from vertex $v_i$. We remark that we could use the solution proposed in [15] on each of the $Q(f,n)$ ranges of distances in which a phrase copy can be found. Unfortunately, this approach would pay another multiplicative factor $\log \sigma$ per symbol and its space complexity would be super-linear in $n$. Conversely, our solution overcomes these drawbacks by deploying two key ideas:

**(1)** The first idea aims at minimizing the space usage by achieving the optimal $O(n)$ working-space bound. It consists of proceeding in $Q(f,n)$ passes, one per interval $I_k$ of possible $d$-costs for the edges in $\widetilde{\mathcal{G}}(S)$. During the $k$th pass, we logically partition the vertices of $\widetilde{\mathcal{G}}(S)$ in blocks of $|I_k|$ contiguous vertices, say $v_{i_k}, v_{i_k+1}, \ldots, v_{i_k+|I_k|-1}$, and compute all $d$-maximal edges which spread out from that block and have copy-distance within $I_k$ (thus they all have the same $d$-cost, say $c(I_k)$). These edges are kept in memory until they are used by `Optimal-Parser`, and discarded as soon as the first vertex of the next block, i.e. $v_{i_k+|I_k|}$, needs to be processed. The next block of $|I_k|$ vertices is then fetched and the process repeats. Actually, all passes are executed in *parallel* to guarantee that all $d$-maximal

edges of $v_i$ are available when processing this vertex. There are $n/|I_k|$ distinct blocks at each pass, and each $d$-maximal edge of a vertex is considered in some pass (because it has $d$-cost in some $I_k$). The space is $\sum_{k=1}^{Q(f,n)} |I_k| = O(n)$ because we keep one $d$-maximal edge per vertex at any pass.
**(2)** The second key idea aims at computing the $d$-maximal edges for that block of $|I_k|$ contiguous vertices in $O(|I_k|)$ time and space. This is what we address below, being the most sophisticated technicality of our solution. As a result, we show that the time complexity of FSG is $\sum_{k=1}^{Q(f,n)}(n/|I_k|)O(|I_k|) = O(n\,Q(f,n))$, i.e., $O(1)$ amortized time per $d$-maximal edge. Combining this fact with the previous observation on the computation of the $\ell$-maximal edges, we get Theorem 8 above.

Let us assume that the alphabet size is $\sigma \leq n$, and consider the $k$th pass of FSG in which we assume that $I_k = [l, r]$. Recall that all distances in $I_k$ can be $f$-encoded in the same number of, say, $c(I_k)$ bits. Let $B = [i, i + |I_k| - 1]$ be the block of (indices of) vertices for which we wish to compute *on-the-fly* the $d$-maximal edges of cost $c(I_k)$. This means that the $d$-maximal edge from vertex $v_h$, $h \in B$, represents a phrase that starts at $S[h]$ and has a copy starting in the *window* $W_h = [h - r, h - l]$. Thus the distance of that copy can be $f$-encoded in $c(I_k)$ bits, and so we will say that the edge has $d$-cost $c(I_k)$. Since this computation must be done for all vertices in $B$, it is useful to consider the window $\mathcal{W}_B = W_i \cup W_{i+|I_k|-1}$ which merges the first and last window and thus spans all positions that can be the (copy-)reference of any $d$-maximal edge outgoing from $B$. Note that $|\mathcal{W}_B| = 2|I_k|$ (see Figure 1 in Appendix C). The next fact is crucial to fast compute all these $d$-maximal edges via indexing data structures built over $S$:

**Fact 9** *If there exists a $d$-maximal edge outgoing from $v_h$ and having $d$-cost $c(I_k)$, then this edge can be found by determining a position $s \in W_h$ whose suffix $S_s$ shares the* maximum *longest common prefix (shortly,* lcp*) with $S_h$.*

**Proof:** Among all positions $s$ in $W_h$ take one whose suffix $S_s$ shares the maximum lcp with $S_h$, and let $q$ be the length of this lcp. Of course, there may exist many such positions, we take just one of them. The edge $(v_h, v_{h+q+1})$ has $d$-cost $c(I_k)$, because $s \in W_h$, and is $d$-maximal because any other position $s' \in W_h$ induces an edge $(v_h, v_{h+q'+1})$ whose length $q' \leq q$, by maximality of $q$. ■

Hereafter we call the position $s$ of Fact 9 *maximal position* for vertex $v_h$, and note that it does exist only if $v_h$ has a $d$-maximal edge of cost $c(I_k)$. Our algorithm will compute the maximal positions of every vertex $v_h$ in $B$ whenever they do exist, otherwise it will assign an *arbitrary* position to $v_h$. The net result is that we will generate a *supergraph* of $\widetilde{\mathcal{G}}(S)$ which is still guaranteed to have the size stated in Lemma 6 and can be created efficiently in $O(|I_k|)$ time and space, as we required above.

Fact 9 has related the computation of maximal positions for the vertices in $B$ to lcp-computations between suffixes in $B$ and suffixes in $\mathcal{W}_B$. Therefore it is natural to resort some indexing data structure, like the compact trie $\mathcal{T}_B$, built over the suffixes of $S$ which start in the range of positions $B \cup \mathcal{W}_B$. Trie $\mathcal{T}_B$ takes $O(|B| + |\mathcal{W}_B|) = O(|I_k|)$ space, and this bound is within our required space complexity. It is not easy to build $\mathcal{T}_B$ in $O(|I_k|)$ time and space, because this time complexity is *independent* of the length of the indexed suffixes and the alphabet size. In Appendix E we prove this result by deploying the fact that the algorithm we detail below does not make any assumption on the edge-ordering of $\mathcal{T}_B$, because it just computes (sort of) lca-queries on its structure.

Given the trie $\mathcal{T}_B$, we notice that the maximal position $s$ for a vertex $v_h$ in $B$ having $d$-cost $c(I_k)$ can be computed by *finding the leaf of $\mathcal{T}_B$ which is labeled with an index $s \in W_h$ and has the deepest lowest common ancestor (shortly,* lca*) with the leaf labeled $h$.* We need to answer this query in $O(1)$ amortized time per vertex $v_h$, since we aim at achieving an $O(|I_k|)$ time complexity over all vertices in $B$. This is not easy because this is *not* the classic lca-query since we do not know $s$, which is actually the position we are searching for! Furthermore, since the leaf $s$ is the closest one to $h$ in

$\mathcal{T}_B$ among the leaves with index in $W_h$, one could think to use proper predecessor/successor queries on a suitable dynamic set of suffixes in $W_h$. Unfortunately, this would take $\omega(1)$ time because of well-known lower bounds [2]. Therefore, in order to answer this query in constant (amortized) time per vertex of $B$, we deploy proper structural properties of the trie $\mathcal{T}_B$ and the problem at hand.

Let $u$ be the `lca` of the leaves labeled $h$ and $s$ in $\mathcal{T}_B$. For simplicity, we assume that the window $W_h$ strictly precedes $B$ and that $s$ is the unique maximal position for $v_h$ (our algorithm deals with these cases too, see the proof of Lemma 10 in Appendix D). We observe that $h$ must be the smallest index that lies in $B$ and labels a leaf descending from $u$ in $\mathcal{T}_B$. In fact assume, by contradiction, that a smaller index $h' < h$ does exist. By definition $h' \in B$ and thus $v_h$ would not have a $d$-maximal edge of $d$-cost $c(I_k)$ because it could copy from the closer $h'$ a possibly longer phrase, instead of copying from the farther set of positions in $W_h$. This observation implies that we have to search only for one maximal position per node $u$ of $\mathcal{T}_B$, and this position refers to the vertex $v_{a(u)}$ whose index $a(u)$ is the smallest one that lies in $B$ and labels a leaf descending from $u$. Computing $a$-values clearly takes $O(|\mathcal{T}_B|) = O(|I_k|)$ time and space via a traversal of the trie $\mathcal{T}_B$.

Now we need to compute the maximal position for $v_{a(u)}$, for each node $u \in \mathcal{T}_B$. We cannot traverse the subtree of $u$ searching for the maximal position for $v_{a(u)}$, because this would take quadratic time complexity overall. Conversely, we define $\mathcal{W}'_B$ and $\mathcal{W}''_B$ to be the first and the second half of $\mathcal{W}_B$, respectively, and observe that any window $W_h$ has its left extreme in $\mathcal{W}'_B$ and its right extreme in $\mathcal{W}''_B$ (see Figure 1 in Appendix C). Therefore the window $W_{a(u)}$ containing the maximal position $s$ for $v_{a(u)}$ overlaps both $\mathcal{W}'_B$ and $\mathcal{W}''_B$. If $s$ does exist for $v_{a(u)}$, then $s$ belongs to either $\mathcal{W}'_B$ or to $\mathcal{W}''_B$, and the leaf labeled $s$ descends from $u$. Hence the maximum (resp. minimum) among the elements in $\mathcal{W}'_B$ (resp. $\mathcal{W}''_B$) that label leaves descending from $u$ must belong to $W_{a(u)}$.

This suggests to compute for each node $u$ the rightmost position in $\mathcal{W}'_B$ and the leftmost position in $\mathcal{W}''_B$ that label a leaf descending from $u$, denoted respectively by $\texttt{max}(u)$ and $\texttt{min}(u)$. This takes $O(|I_k|)$ time with a post-order visit of $\mathcal{T}_B$. We can now efficiently compute $\texttt{mp}[h]$ as the maximal position for $v_h$, if it exists, or otherwise set $\texttt{mp}[h]$ arbitrarily. We initially set all $\texttt{mp}$'s entries to `nil`; then we visit $\mathcal{T}_B$ in post-order and perform, at each node $u$, the following two checks whenever $\texttt{mp}[a(u)] = \texttt{nil}$: If $\texttt{min}(u) \in W_{a(u)}$, we set $\texttt{mp}[a(u)] = \texttt{min}(u)$; if $\texttt{max}(u) \in W_{a(u)}$, we set $\texttt{mp}[a(u)] = \texttt{max}(u)$. At the end of the visit, if $\texttt{mp}[a(u)]$ is still `nil` we set $\texttt{mp}[a(u)] = a(\texttt{parent}(u))$ whenever $a(u) \neq a(\texttt{parent}(u))$. This last check is needed (see proof of Lemma 10 in Appendix D) to manage the case in which $S[a(u)]$ can copy the phrase starting at its position from position $a(\texttt{parent}(u))$ and, additionally, we have that $B$ overlaps $\mathcal{W}_B$ (which may occur depending on $f$). Since $\mathcal{T}_B$ has size $O(|I_k|)$, the overall algorithm requires $O(|I_k|)$ time and space in the worst case, and hence Theorem 8 follows. Correctness follows from lemma below (proof in Appendix D).

**Lemma 10** *For each position $h \in B$, if there exists a $d$-maximal edge outgoing from $v_h$ and having $d$-cost $c(I_k)$, then* $\texttt{mp}[h]$ *is equal to its maximal position.*

## 6 Conclusions

Our bit-optimal parsing scheme can be extended to variants of `LZ77` which deploy parsers that refer to a *bounded* compression-window (the typical scenario of `gzip` and its derivatives [30]). In this case, `LZ77` selects the next phrase by looking only at the most recent $M$ input symbols. Since $M$ is usually a constant of few Kbs [30], the running time of our algorithm is $O(n\, Q(g, n))$, given that $Q(f, M)$ turns out to be a constant. This complexity could be further refined as $O(n\, Q(g, \ell))$ by considering the length $\ell$ of the *longest repeated substring* in $S$. Furthermore, if $S$ is generated by an ergodic source [33] and $g$ is taken to be the classic Elias' code, we have $Q(g, \ell) = O(\log \log n)$ so that the complexity of our algorithm results $O(n \log \log n)$ time and $O(n)$ space for this class of strings.

We finally notice that, although we have mainly dealt with the LZ77-dictionary, the techniques presented in this paper can be extended to design efficient bit-optimal compressors for other on-line dictionary construction schemes, like LZ78 (details in the full paper). The main open question we leave with this paper is to extend our results to *statistical* encoding functions like Huffman or Arithmetic coders applied on the integral range $1 \dots n$ [36]. They do not necessarily satisfy Property 1 because it might be the case that $|f(x)| > |f(y)|$, whenever the integer $y$ occurs more frequently than the integer $x$ in the parsing of $S$. We argue that it is not trivial to design a bit-optimal compressor for these encoding functions because their codeword lengths change as it changes the set of distances and lengths used in the parsing process.

## References

[1] A. Amir, G. M. Landau, and E. Ukkonen. Online timestamped text indexing. *Inf. Process. Lett.*, 82(5):253–259, 2002.

[2] P. Beame and F. E. Fich. Optimal bounds for the predecessor problem. In *Proceedings of the 31st ACM Symposium on Theory of Computing*, pages 295–304, 1999.

[3] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[4] J. Bksi, G. Galambos, U. Pferschy, and G.J. Woeginger. Greedy algorithms for on-line data compression. *Journal of Algorithms*, 25(2):274–289, 1997.

[5] R. Cilibrasi and P. M. B. Vitányi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, 2005.

[6] M. Cohn and R. Khazan. Parsing with prefix and suffix dictionaries. In *Data Compression Conference*, pages 180–189, 1996.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* The MIT Press and McGraw-Hill Book Company, 2001.

[8] G. Cormode and S. Muthukrishnan. Substring compression problems. In *SODA*, pages 321–330, 2005.

[9] M. Crochemore, L. Ilie, and W. F. Smyth. A simple algorithm for computing the Lempel-Ziv factorization. In *DCC*, pages 482–488, 2008.

[10] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

[11] P. Fenwick. Universal codes. In *Lossless Compression Handbook*, pages 55–78. Academic Press, 2003.

[12] P. Ferragina, R. Giancarlo, and G. Manzini. The engineering of a compression boosting library: Theory vs practice in bwt compression. In *ESA*, pages 756–767, 2006.

[13] P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, 52:688–713, 2005.

[14] P. Ferragina and G. Navarro. Site pizza&chili (`http://pizzachili.di.unipi.it` or `http://pizzachili.dcc.uchile.cl`).

[15] E. R. Fiala and D. H. Greene. Data compression with finite windows. *Commun. ACM*, 32(4):490–505, 1989.

[16] A. Langiu G. Della Penna, F. Mignosi and A. Ulisse. On dictionary-symbolwise data compression. In `http://www.di.univaq.it/%7mignosi/ulicompressor.php`, 2006.

[17] T. Gagie and G. Manzini. Space-conscious compression. In *MFCS*, pages 206–217, 2007.

[18] R. F. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. *ACM Transactions on Algorithms*, 2(4):510–534, 2006.

[19] Gzip home page. `http://www.gzip.org`.

[20] J. Katajainen and T. Raita. An approximation algorithm for space-optimal encoding of a text. *Computer Journal*, 32(3):228–237, 1989.

[21] J. Katajainen and T. Raita. An analysis of the longest match and the greedy heuristics in text encoding. *Journal of the ACM*, 39(2):281–294, 1992.

[22] S. T. Klein. Efficient optimal recompression. *Computer Journal*, 40(2/3):117–126, 1997.

[23] R. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel–Ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 1999.

[24] M. Lewenstein, V., Mäkinen, and S. J. Puglisi. Personal communications.

[25] Y. Matias and S.C. Şahinalp. On the optimality of parsing in dynamic dictionary based data compression. In *SODA*, pages 943–944, 1999.

[26] E. M. McCreight. Priority search trees. *SIAM J. Comput.*, 14(2):257–276, 1985.

[27] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), 2007.

[28] S. J. Puglisi, W. F. Smyth, and A. H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2), 2007.

[29] N. Rajpoot and C. Sahinalp. *Handbook of Lossless Data Compression*, chapter Dictionary-based data compression, pages 153–167. Academic Press, 2002.

[30] D. Salomon. *Data Compression: the Complete Reference, 3rd Edition*. Springer Verlag, 2004.

[31] E. J. Schuegraf and H. S. Heaps. A comparison of algorithms for data base compression by use of fragments as language elements. *Information Storage and Retrieval*, 10(9-10):309319, 1974.

[32] M. E. Gonzalez Smith and J. A. Storer. Parallel algorithms for data compression. *Journal of the ACM*, 32(2):344–373, 1985.

[33] W. Szpankowski. Asymptotic properties of data compression and suffix trees. *IEEE Transactions on Information Theory*, 39(5):1647–1659, 1993.

[34] J. S. Vitter and P. Krishnan. Optimal prefetching via data compression. *Journal of the ACM*, 43(5):771–793, 1996.

[35] D. E. Willard. Examining computational geometry, van emde boas trees, and hashing from the perspective of the fusion tree. *SIAM J. Comput.*, 29(3), 2000.

[36] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, second edition, 1999.

[37] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transaction on Information Theory*, 23:337–343, 1977.

[38] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

[39] Jacob Ziv. Classification with finite memory revisited. *IEEE Transactions on Information Theory*, 53(12):4413–4421, 2007.

## A On the greedy LZ77-parsing with rightmost copies

We make use of a slightly modified version of the tree covering procedure of [18] on the internal nodes of $\mathcal{ST}_\mathcal{C}$. Given $\mathcal{ST}_\mathcal{C}$ and an integer parameter $P \geq 2$, in our case $P = \sigma$, this procedure covers the internal nodes of $\mathcal{ST}_\mathcal{C}$ in a number of connected subtrees, all of which have size $\Theta(P)$, except the one which contains the root of $\mathcal{ST}_\mathcal{C}$ that has size $O(P)$. Two of these subtrees are either disjoint or intersect at their common root. We refer to Section 2 of [18] for more details. In our modification we impose that there is no node in common to two subtrees, because we move their common root to the subtree that contains its parent. None of the above properties change, except for the fact that each cover could now be a subforest instead of subtree of $\mathcal{ST}_\mathcal{C}$. Let $F_1, F_2, \ldots F_t$ be the subforests obtained by the above covering. Clearly, we have that $t = O(k/P)$ where $k$ is the number of internal nodes in $\mathcal{ST}_\mathcal{C}$.

We define the tree $\mathcal{ST}_\mathcal{SC}$ whose leaves are the leaves of $\mathcal{ST}_\mathcal{C}$ and whose internal nodes are the above subforests. With a little abuse of notation, let us refer with $F_i$ to the node in $\mathcal{ST}_\mathcal{SC}$ corresponding to the subforest $F_i$. The leaf $l$ having $u$ as parent in $\mathcal{ST}_\mathcal{C}$, is thus connected to the node $F_i$ in $\mathcal{ST}_\mathcal{SC}$, where $F_i$ is the forest that contains the node $u$. Notice that roots of subtrees in any subforest $F_i$ have common parent in $\mathcal{ST}_\mathcal{C}$.

The computation of the rightmost copy for a phrase $p_i$ is now divided in two phases (see Section 3 for the notation and the underlying main algorithm). Let $F_i$ be the subforest that contains $u_{p_i}$, the node spelled out by the phrase starting at $S[p_i]$. In the first phase, we compute the rightmost copy for the phrase starting at $p_i$ among the descendants of $u_{p_i}$ in $\mathcal{ST}_\mathcal{SC}$ that belong to subforests different from $F_i$. In the second phase, we compute its rightmost copy among the descendants of $u_{p_i}$ in $F_i$. The maximum between these two values will give the rightmost copy for $p_i$, of course. To solve the former problem, we execute the algorithm of Section 3 on $\mathcal{ST}_\mathcal{SC}$. It simply visits all subforests descendant from $F_i$ in $\mathcal{ST}_\mathcal{SC}$, each of them maintaining the rightmost position among its already scanned leaves, and returns the maximum of these value. Since groups of $\sigma$ nodes of $\mathcal{ST}_\mathcal{C}$ have single nodes in $\mathcal{ST}_\mathcal{SC}$, the algorithm requires $O(n)$ time.

The latter problem is solved with a new algorithm exploiting the fact that the number of nodes in $F_i$ is $O(\sigma)$ and resorting to dynamic Range Maximum Queries (RMQ) on properly defined arrays [26]. To be precise, we assign to each node of $F_i$ an unique identifier in $[m]$ that corresponds to the time of its visit in a depth-first traversal of $F_i$. Notice that the nodes in the subtree rooted at some node $u$ receive integers spanning the whole range from the starting time to the ending time of the DFS-visit of $u$. We use an array $A_{F_i}$ that has an entry for each node of $F_i$. Initially, all entries are set to $-\infty$. The entry corresponding to any node has index equal to its time visit. We build on each array $A_{F_i}$ a dynamic data structure that answers range maximum queries. For this purpose we use a simple balanced tree augmented with the maximum of the descending leaves in each of its nodes. This way Range-Max queries and updates on $A_{F_i}$ take $O(\log \sigma)$ time in the worst case.

Now, we proceed as in the algorithm of Section 3, and process string $S$ from left to right. When a position $j$ of $S$ is processed, we identify the subforest $F_i$ containing the father of the leaf labeled $j$ in $\mathcal{ST}_\mathcal{C}$ and we set to $j$ the corresponding entry in $A_{F_i}$ (this induces a change in the underlying RMQ data structure). If $j$ is the starting position of a phrase, we identify the subforest $F_x$ containing the node $u_j$ and compute its rightmost copy in $F_x$, by resorting to a RMQ on $A_{F_x}$. The left and right indexes for the range query are, respectively, the starting and ending time of the visit of $u_j$ in $F_x$.

It is easy to notice that the overall complexity of this algorithm is dominated by the $O(n)$ updates to the RMQ data structures and the $O(k)$ queries onto them (recall that $k$ is the number of phrases of the LZ77-greedy parsing of $S$). Our algorithm then takes $O(n \log \sigma)$ time and $O(n)$ space. A further improvement can be obtained by adopting an idea similar to the one in [35][Section 5] to reduce the height of that balanced tree and, consequently, our time complexity by a factor $O(\log \log n)$. This proves Lemma 1.

## B Proof of Fact 4

**Fact** 4. *Given a vertex $v_i$, for any pair of vertices $v_{j'}, v_{j''} \in FS(v_i)$ such that $j' < j''$, we have that $c(v_i, v_{j'}) \leq c(v_i, v_{j''})$. The same property holds for $v_{j'}, v_{j''} \in BS(v_i)$.*

**Proof:** We have that $d_{i,j'} \leq d_{i,j''}$ and $\ell_{i,j'} < \ell_{i,j''}$ because $S[i : j' - 1]$ is a prefix of $S[i : j'' - 1]$ and thus the first substring occurs wherever the latter occurs. The property holds because $f$ and $g$ satisfy the Increasing Cost Property 1. ∎

## C An illustrative picture

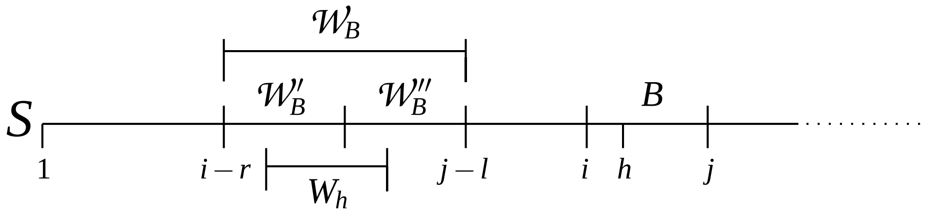

Figure 1: Interval $B = [i, j]$ with $j = i + |I_k| - 1$, window $\mathcal{W}_B$ and its two halves $\mathcal{W}'_B, \mathcal{W}''_B$.

## D Proof of Lemma 10

Recall that, by definition, $a(u)$ is the smallest index that lies in block $B$ and labels a leaf descending from node $u$ in $\mathcal{T}_B$.

**Lemma** 10. *For each position $h \in B$, if there exists a d-maximal edge outgoing from $v_h$ and having d-cost $c(I_k)$, then* $\texttt{mp}[h]$ *is equal to its maximal position.*

**Proof:** Recall that $B = [i, i+|I_k|-1]$ and consider the longest path $\pi = u_1 u_2 \dots u_z$ in $\mathcal{T}_B$ that starts from the leaf $u_1$ labeled with $h \in B$ and goes upward until the traversed nodes satisfy the condition $a(u_j) = h$, here $j = 1, \dots, z$. By definition of $a$-value (see above), we know that all leaves descending from $u_z$ and occurring in $B$ are labeled with an index which is larger than $h$. Clearly, if $\texttt{parent}(u_z)$ does exist, then it is $a(\texttt{parent}(u_z)) < h$. There are two cases for the final value stored in $\texttt{mp}[h]$.

**Case 1.** Suppose that $\texttt{mp}[h] \in W_h$. We want to prove that $\texttt{mp}[h]$ is the index of the leaf which has the deepest $\texttt{lca}$ with $h$ among all the other leaves labeled with an index in $W_h$. Let $u_x \in \pi$ be the node in which the value of $\texttt{mp}[h]$ is assigned. Then, by our algorithm it is $a(u_x) = h$. Assume now that there exists at least another index in $W_h$ whose leaf has a deeper $\texttt{lca}$ with leaf $h$. This $\texttt{lca}$ must lie on $u_1 \dots u_{x-1}$, say $u_l$. Since $W_h$ is a window having its left extreme in $\mathcal{W}'_B$ and its right extreme in $\mathcal{W}''_B$, the value $\texttt{max}(u_l)$ or $\texttt{min}(u_l)$ must lie in $W_h$ and thus the algorithm has set $\texttt{mp}[h]$ to one of these positions, because of the post-order visit of $\mathcal{T}_B$. Therefore $\texttt{mp}[h]$ must be the index of the leaf having the deepest $\texttt{lca}$ with $h$, and thus by Fact 9 it is its maximal position.

**Case 2.** Suppose that $\texttt{mp}[h] \notin W_h$ and, thus, it cannot be a maximal position for $v_h$. We have to prove that it does not exist a $d$-maximal edge outgoing from the vertex $v_h$ with cost $c(I_k)$. Let $S_s$ be the suffix in $W_h$ having the maximum $\texttt{lcp}$ with $S_h$, and let $l$ be the $\texttt{lcp}$-length. Values $\texttt{min}(u_i)$ and $\texttt{max}(u_i)$ do not belong to $W_h$, for any node $u_i \in \pi$ with $a(u_i) = h$, otherwise $\texttt{mp}[h]$ would have been assigned with an index in $W_h$ (contradicting the hypothesis). Thus the value of `mp`$[h]$ remains `nil` up to node $u_z$. This implies that no suffix descending from $u_z$ starts in $W_h$ and, in particular, $S_s$ does not descend from $u_z$. Therefore, the `lca` between leaves $h$ and $s$ is a node in the path from `parent`$(u_z)$ to the root of $\mathcal{T}_B$, and the $\texttt{lcp}(S_{a(\texttt{parent}(u_z))}, S_h) \geq \texttt{lcp}(S_s, S_h) = l$. Since $a(\texttt{parent}(u_z)) < a(u_z)$ and belongs to $B$, this position is nearer to $h$ than any other position in $W_h$, and shares a longer prefix with $S_h$. So we found a longer edge from $v_h$ with smaller $d$-cost. This way $v_h$ has no $d$-maximal edge of cost $c(I_k)$ in $\widetilde{\mathcal{G}}(S)$. ∎

## E Building $\mathcal{T}_B$ optimally

We show how to build $\mathcal{T}_B$ in $O(|I_k|)$ time and space, thus within a time complexity which is *independent* of the length of the indexed suffixes and the alphabet size. To achieve this result we deploy the crucial fact that the algorithm of Section 5.2 does not make any assumption on the ordering of the edges in $\mathcal{T}_B$, because it just computes (sort of) `lca`-queries on its structure.

At preprocessing time we build the suffix array of the whole string $S$ and a data structure that answers constant-time `lcp`-queries between pair of suffixes (see e.g. [28]). These data structures can be built in $O(n)$ time and space, when $\sigma = O(n)$. For larger alphabets, we need to add $T_{sort}(n, \sigma)$ time, which takes into account the cost of sorting the symbols of $S$ and re-mapping them to $[n]$ (see Theorem 8).

Let us first assume that $B$ and $\mathcal{W}_B$ are contiguous and form the range $[i, i + 3|I_k| - 1]$. If we had the sorted sequence of suffixes starting in $S[i, i + 3|I_k| - 1]$, we could easily build $\mathcal{T}_B$ in $O(|I_k|)$ time and space by deploying the above `lcp`-data structure. Unfortunately, it is unclear how to obtain from the suffix array of the whole $S$, the sorted sub-sequence of suffixes *starting* in the range $[i, i + 3|I_k| - 1]$ by taking $O(|B| + |\mathcal{W}_B|) = O(|I_k|)$ time (notice that these suffixes have length $\Theta(n - i)$). We cannot perform a sequence of predecessor/successor queries because they would take $\omega(1)$ time each [2]. Conversely, we resort the key observation above that $\mathcal{T}_B$ does not need to be ordered, and thus devise a solution which builds an *unordered* $\mathcal{T}_B$ in $O(|I_k|)$ time and space, passing through the construction of the suffix array of a *transformed* string. The transformation is simple. We first map the distinct symbols of $S[i, i+3|I_k|-1]$ to the first $O(|I_k|)$ integers. This mapping *does not need to* reflect their lexicographic order, and thus can be computed in $O(|I_k|)$ time by a simple scan of those symbols and the use of a table $T$ of size $\sigma < n$. Then, we define $S^T$ as the string $S$ which has been transformed by re-mapping some of the symbols according to table $T$ (namely, those occurring in $S[i, i + 3|I_k| - 1]$). We can prove that

**Lemma 11** *Let $S_i, \ldots, S_j$ be a contiguous sequence of suffixes in $S$. The re-mapped suffixes $S_i^T \ldots S_j^T$ can be lexicographically sorted in $O(j - i + 1)$ time.*

**Proof:** Consider the string of pairs $w = \langle S^T[i], b_i\rangle \ldots \langle S^T[j], b_j\rangle\$$, where $b_h$ is 1 if $S^T_{h+1} > S^T_{j+1}$, $-1$ if $S^T_{h+1} < S^T_{j+1}$, or 0 if $h = j$. The ordering of the pairs is defined component-wise, and we assume that \$ is a special "pair" larger than any other pair in $w$. For any pair of indices $p, q \in [1 \ldots j - i]$, it is $S^T_{p+i} > S^T_{q+i}$ iff $w_p > w_q$. In fact, suppose that $w_p > w_q$ and set $r = \texttt{lcp}(w_p, w_q)$. We have that $w[p + r] = \langle S^T[p + i + r], b_{p+i+r}\rangle > \langle S^T[q + i + r], b_{q+i+r}\rangle = w[q + i + r]$. Hence $S^T_{p+i+r} > S^T_{q+i+r}$, by definition of the $b$'s. Therefore $S^T_{p+i} > S^T_{q+i}$, since their first $r$ symbols are equal. This implies that sorting the suffixes $S^T_i, \ldots, S^T_j$ reduces to computing the suffix array of $w$, and this takes $O(|w|)$ time given that the alphabet size is $O(|w|)$ [28]. Clearly, $w$ can be constructed in that time bound because comparing $S^T_z$ with $S^T_{j+1}$ takes $O(1)$ time via an `lcp`-query on $S$ (using the proper data structure above) and a check at their first mismatch. ∎

Lemma 11 allows us to generate the compact trie of $S_i^T, \dots, S_{i+3|I_k|-1}^T$, which is equal to the (unordered) compacted trie of $S_i, \dots, S_{i+3|I_k|-1}$ after replacing every `ID` assigned by table $T$ with its original symbol in $S$. We finally notice that if $B$ and $\mathcal{W}_B$ are not contiguous (as instead we assumed above), we can use a similar strategy to sort separately the suffixes in $B$ and the suffixes in $\mathcal{W}_B$, and then merge these two sequences together by deploying the `lcp`-data structure mentioned at the beginning of this section.