ON THE BIT-COMPLEXITY OF LEMPEL-ZIV COMPRESSION*

PAOLO FERRAGINA†, IGOR NITTO‡, AND ROSSANO VENTURINI†

Abstract. One of the most famous and investigated lossless data-compression schemes is the one introduced by Lempel and Ziv about 30 years ago [38]. 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, *"we are not aware of any parsing scheme that achieves optimality [...] 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 some issues pertaining to the bit-complexity of LZ77-based compressors, the most powerful variant of the LZ-compression scheme, and we design algorithms which achieve bit-optimality in the compressed output size by taking efficient/optimal time and optimal space.

Key words. compression, Lempel-Ziv 1977, dynamic programming

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`, `7zip`, `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 phrases in a dictionary. 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 [38, 39], and all their variants [30], are interesting examples of *dynamic* dictionary-based compressors.

Many theoretical and experimental results have been dedicated to LZ-compressors in these thirty years 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 [7] or classification tools [37], in designing *optimal* pre-fetching mechanisms [34], and in streaming or on-the-fly compression applications [10, 19].

In this paper we address some key issues which arise when dealing with the output-size in bits of the so called LZ77*-parsing scheme*, namely the one in which 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),

*This work is an extended version of the paper appeared in the *Proceedings of the 20th ACM-SIAM Symposium on Algorithms*, pp. 768-777, 2009. This work was partially supported by MIUR of Italy under projects PRIN ARS Technomedia 2012 and FIRB Linguistica 2006, the Midas EU Project, and the eCloud EU Project.

†Dipartimento di Informatica, University of Pisa

‡Microsoft STC, London

and phrase-codewords consist of triples $\langle d, \ell, c\rangle$ where d is the relative *offset* of the copied phrase ($d \leq M$), ℓ is the length of the phrase and c is the single (new) character following it. Classically, the LZ77-parser adopts a *greedy* rule, namely one that at each step takes the *longest* dictionary phrase which is a prefix of the currently unparsed suffix of the input string. This greedy parsing can be computed in $O(n \log \sigma)$ time and $O(M)$ space [18].[1] The greedy parsing is optimal with respect to the *number of phrases* in which S can be parsed by any suffix-complete dictionary (like the LZ77-dictionary). Of course, the number of parsed phrases influences the compression ratio and, indeed, various authors [38, 24] proved that greedy parsing achieves asymptotically the *(empirical) entropy* of the source generating the input string S. But these fundamental results have *not 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 we assume that the phrases are compressed via an *equal-length* encoder, like in [24, 30, 38], then the output produced by the greedy parsing scheme is *bit optimal*. But if one aims for higher compression, *variable-length encoders* should be taken into account (see e.g. [36, 13], and the software `gzip` [2]), 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 that deploy the LZ77-dictionary built on an unbounded window (namely, it is $M = n$). Our results extend easily to windows of arbitrary size $M < n$, as we will comment in Section 7.

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 number of bits output by the final compressor, the greedy parser should obviously select the *closest* copy of each phrase w_i in S, and thus the smallest possible d_i. Surprisingly enough, known implementations of greedy parsers are time optimal but not bit-optimal, because they select an arbitrary or the leftmost occurrence of the longest copied phrase (see [11] and references therein), or they select the closest copy but take $O(n \log n)$ suboptimal time [1, 25]. 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 $O\left(n\left(1 + \frac{\log \sigma}{\log \log n}\right)\right)$ overall time and $O(n)$ space (Theorem 3.1).

Problem 2. How good is the greedy LZ77-parsing of S whenever the compression cost is measured in terms of number of bits produced in output? We show that the greedy selection of the longest dictionary phrase at each parsing step is not optimal, and this may be larger than the bit-optimal parsing by a multiplicative factor $\Omega(\log n / \log \log n)$, which is unbounded asymptotically (Section 4). Additionally, we show that this lower-bound is tight up to a factor $\Theta(\log \log n)$, and we support these theoretical figures with some experimental results which stress the practical importance of finding the bit-optimal parsing of S.

[1] Recently, [11] 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., $M = n$.

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

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], in that they take $\Theta(n^2)$ worst-case time and space, or they are approximate [21], or they rely on heuristics [23, 32, 6, 8] 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 (possibly, optimal) and space optimal for the problem above (Theorem 5.4).

Technically speaking, we follow [31] and model the search for a bit-optimal 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 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 greater 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 [13] and `gzip`), 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 5.3). Additionally, we design an algorithm that solves the SSSP on the subgraph $\widetilde{\mathcal{G}}(S)$ without materializing it *all at once*, but creating and exploring 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 $|\widetilde{\mathcal{G}}(S)|$. This way, the compressor is optimal in the size of the sub-graph which is $O(n \log n)$ for a large class of integer encoders, like Elias, Rice, and Fibonacci codes [36, 13], and it is optimal $O(n)$ for (most of) the encodings used by `gzip`. This is the first result providing a *positive answer* to Rajpoot-Sahinalp's question above.

Problem 4. How much efficient is *in practice* our bit-optimal LZ77-compressor? To establish this, we have taken several freely available text collections, and compared our compressor against the classic `gzip` and `bzip2`[3], as well as against the state-of-the-art *boosting* compressor of [16, 15]. Section 6 reports some experimental figures, and comments on our theoretical findings as well as on possible algorithm-engineering research directions which deserve further attention.

Map of the paper. In Section 2 we introduce the notation and basic terminology. Section 3 will design a bit-optimal parser based on the LZ77-dictionary and the greedy-parsing rule which efficiently parses the input string S in $O\left(n\left(1+\frac{\log\sigma}{\log\log n}\right)\right)$ time and $O(n)$ space (Theorem 3.1). Section 4 proposes an infinite class of strings for which the compression gap between the parsing based on the greedy-rule and the

[3] Bzip2 home page `http://www.bzip.org/`.

fully bit-optimal parsing is unbounded in the general case of variable-length integer encoders (Lemma 4.1). Starting from this negative result, we design in Section 5.2 our novel bit-optimal parsing strategy which is efficient (possibly optimal) both in time and space (Theorem 5.4). Finally Section 6 will report some experimental comparisons among our novel LZ77-based compressor and some well known compression tools, such as `gzip` and `bzip2`, thus showing the effectiveness of our approach and its promising performance (Table 6). Concluding remarks and possible future directions of research will be discussed in Section 7.

2. Notation and terminology. Let $S[1,n]$ be a string drawn from an alphabet $\Sigma = [\sigma]$. In the following we will assume that σ is at most n. [4] We use $S[i]$ to denote the ith symbol of S; $S[i:j]$ to denote the substring (also called the *phrase*) extending from the ith to the jth 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 concentrate on LZ77-compression with an unbounded window size, so we will drop the specification "LZ77" unless this will be required to make things unambiguous. The compressor, as any dictionary-based compressor, will work 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 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* (hereafter assumed unbounded, for simplicity). The classic parsing-rule adopted by most LZ77-compressors 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 is usually 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 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 ℓ_i is its length $|w_i|$. Every first occurrence of a new symbol c is encoded as $\langle 0, c \rangle$. We allow self-referencing phrases, i.e., a phrase's source could overlap the phrase itself.

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 greater integers:

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

This property is satisfied by most known integer encoders— like equal-length codewords, Elias codes [36], Rice's codes [30], Fibonacci's codes [13]— which are used to design data compressors [30], compressed full-text indexes [27] and search engines [36].

[4] In case of a larger alphabet, our algorithms are still correct but we need to add the term $T_{sort}(n, \sigma)$ to their time complexities, which denotes the time required to sort/remap all distinct symbols of S into the range $[n]$.

3. An efficient and bit-optimal greedy parsing. In this section we describe how to compute efficiently the *greedy* parsing that minimizes the final compressed size. We remark that the minimization here is done with respect to all the LZ77-parsings that follow the greedy strategy for selecting their phrases; this means that these parsings are constrained to take at every step the longest possible phrase matching the current suffix, but are free to choose which previous copy of this phrase to select.

Let f and g be two integer encoders which satisfy the Increasing Cost Property (possibly $f = g$). We denote by $\texttt{LZ}_{f,g}(S)$ 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 ℓ_i of parsed phrase w_i. Thus, in $\texttt{LZ}_{f,g}(S)$ any phrase w_i is encoded in $|f(d_i)| + |g(\ell_i)|$ bits. Given that the parsing is the greedy one, ℓ_i is fixed (being the length of the longest copy), so we minimize $|\texttt{LZ}_{f,g}(S)|$ 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 $|\texttt{LZ}_{f,g}(S)|$ 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 $O(n(1 + \log \sigma / \log\log n))$ time. This algorithm is the fastest known in the literature [11]. It requires the suffix tree $\mathcal{ST}$ of S and the parsing of S which consists of, say, $k \leq n$ phrases. It is well known that all these machineries can be computed in linear time and space. 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 σ.

All marked nodes can be computed in $O(n)$ time by searching each phrase in the suffix tree $\mathcal{ST}$. 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 easy to see that $\mathcal{ST}_\mathcal{C}$ consists of $O(k)$ internal nodes and n leaves, and that it can be built in $O(n)$ time via a top-down visit of $\mathcal{ST}$.

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 parent of the leaf labeled j in $\mathcal{ST}_\mathcal{C}$. The key point now 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 internal 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 internal nodes in the subtree rooted at u_{p_i} in $\mathcal{ST}_\mathcal{C}$. In

fact, by construction, there can be at most σ repetitions of the same phrase in the 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 properties 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 $\ell_i = O(|w_i|)$. Summing up on all nodes, we get $\sum_{i=1}^{k} O(|w_i|) = O(n)$. Thus, the above algorithm requires $O(\sigma \times n)$ time, which is linear whenever $\sigma = O(1)$.

Now we will show how to further reduce the time complexity to $O\left(n\left(1 + \frac{\log \sigma}{\log \log n}\right)\right)$ by properly combining a slightly modified variant of the tree covering procedure of [20] with a dynamic Range Maximum Query data structure [26, 35] applied on properly composed arrays of integers. Notice that this improvement leads to an algorithm requiring $O(n)$ time for alphabets of size poly-logarithmic in n.

Given $\mathcal{ST}_\mathcal{C}$ and an integer parameter $P \geq 2$ (in our case $P = \sigma$) this procedure covers the k 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 $O(P)$. Any two of these subtrees are either disjoint or intersect at their common root. (We refer to Section 2 of [20] 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, where we clearly have that $t = O(k/P)$.

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. 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 our previous algorithm 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 $P = \sigma$ nodes of $\mathcal{ST}_\mathcal{C}$ have single nodes in $\mathcal{ST}_{\mathcal{SC}}$, in this case our previous 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]. We assign to each node of F_i an unique integer in the range $[|F_i|]$ that corresponds to the time of its visit in a depth-first traversal of F_i. This way the nodes in the subtree rooted at some node u are identified by 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 at the position specified by its starting-time of the DFS-visit. Initially, all entries are set to $-\infty$; as algorithm proceeds, nodes/entries of F_i/A_{F_i} will get values that denote the

rightmost position of the copies of their corresponding phrases in S. Precisely, as done before S is scanned rightward and 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 change to j the entry in A_{F_i} corresponding to that node. If j is also the starting position of a phrase, we identify the subforest F_x containing the node u_j which spells out that phrase (it is an ancestor of the leaf labeled j)[5] and compute its rightmost copy in F_x by executing 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.

Since A_{F_i} is changed as S is processed, we need to solve a dynamic Range Maximum Queries. To do that we build on each array A_{F_i} a binary balanced tree in which leaves correspond to entries of the array while each internal node stores the maximum value in its subtree (hence, sub-range of A_{F_i}). In this way, Range-Max queries and updates on A_{F_i} take $O(\log \sigma)$ time in the worst case. Indeed, an update may possibly change the values stored in the nodes of a leaf-to-root path having $O(\log \sigma)$ length; a Range-Max query has to compute the maximum among the values stored in (at most) $O(\log \sigma)$ nodes, these nodes are the ones covering the queried interval.

We notice that the overall complexity of the algorithm is dominated by the $O(n)$ updates to the RMQ data structures (one for each text position) and the $O(k)$ RMQ queries (one for each phrase). Then the algorithm takes $O(n \log \sigma + k \log \sigma) = O(n \log \sigma)$ 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 each balanced tree and, consequently, our time complexity by a factor $O(\log \log n)$. The idea is to replace each of the above binary balanced trees with a balanced tree with $(1/2) \log n / \log \log n$ branching factor. The children of each node u in this tree are associated with the maximum value stored in their subtrees. Together with these maxima, we store in u an array with an entry of $\log \log n$ bits for each child. The i-th entry stores the rank of the maximum associated with the i-th child with respect to the maxima of the other children. The above array fits in $(1/2) \log n$ bits. Given a range of children, a query asks to compute in constant time the position of the maximum associated with the children in this range. The query is solved with a lookup in a table of size $O(\sqrt{n} \log^2 n \log \log n)$ bits which tabulates the result of any possible query on any possible array of ranks. Instead, an update asks to increase the maximum associated with a child and adjusts accordingly the array of ranks. For this aim we resort to a table of size $O(\sqrt{n} \log^2 n)$ bits which tabulates the result of any possible update on any possible array of ranks.

THEOREM 3.1. *Given a string $S[1, n]$ drawn from an alphabet $\Sigma = [\sigma]$, there exists an algorithm that computes the greedy parsing of S and reports the rightmost copy of each phrase in the* LZ77*-dictionary taking $O\left(n\left(1 + \frac{\log \sigma}{\log \log n}\right)\right)$ time and $O(n)$ space.*

4. On the bit-efficiency of the greedy LZ77-parsing. We have already noticed in the Introduction that the greedy strategy used by $\mathtt{LZ}_{f,g}(S)$ is not necessarily bit-optimal, so we will hereafter use $\mathsf{OPT}_{f,g}(S)$ to denote the *Bit-Optimal* LZ77*-parsing* of S relative to the integer encoding functions f and g. $\mathsf{OPT}_{f,g}(S)$ computes a parsing of S which uses phrases extracted from the LZ77-dictionary, encodes these phrases by using f and g, and *minimizes* the total number of bits in output. Of course $|\mathtt{LZ}_{f,g}(S)| \geq |\mathsf{OPT}_{f,g}(S)|$, but we aim at establishing how much worse the greedy pars-

[5] Notice that the node u_j can be identified during the rightward scanning of S as usually done in LZ77-parsing, taking $O(n)$ time for all identified phrases.

ing can be with respect to the bit-optimal one. In what follows we identify an infinite family of strings S for which $\frac{|\mathtt{LZ}_{f,g}(S)|}{|\mathsf{OPT}_{f,g}(S)|} = \Omega\left(\frac{\log n}{\log\log n}\right)$, 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', Rice's, and Fibonacci's coders belong to this family). These encodings satisfy the increasing cost property stated in section 2. So taking inspiration from the proof of Lemma 4.2 in [24], 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 integer l. The greedy LZ77-parser partitions S_l as[6]:

$$(b)\ (a)\ (a^{l-1})\ (c)\ (c^{2^l-1})\ (ba)\ (ba^2)\ (ba^3)\ \ldots\ (ba^l),$$

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 $l \times |f(2^l)| = \Theta(l^2)$ bits, since we are counting only the cost for encoding the distances.

A more parsimonious parser 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 the encoding of this parsing, called $\mathsf{rOPT}(S_l)$, takes $|g(2^l-1)| + |g(l-1)| + \sum_{i=2}^{l}[|f(i)| + |g(i)| + |f(0)|] + O(l) = O(l \log l)$ bits.

LEMMA 4.1. *There exists an infinite family of strings such that, for any of its elements S, it is $|\mathtt{LZ}_{f,g}(S)| \geq \Theta(\log|S|/\log\log|S|)\,|\mathsf{OPT}_{f,g}(S)|$.*

Proof. Since $\mathsf{OPT}(S_l)$ is a parsimonious parser, not necessarily the optimal one, it is $|\mathsf{OPT}(S_l)| \leq |\mathsf{rOPT}(S_l)|$. Then we can write: $\frac{|\mathtt{LZ}_{f,g}(S_l)|}{|\mathsf{OPT}_{f,g}(S_l)|} \geq \frac{|\mathtt{LZ}_{f,g}(S_l)|}{|\mathsf{rOPT}(S_l)|} \geq \Theta\left(\frac{l}{\log l}\right)$. 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 6 will show that this gap is not negligible in practice too.

Additionally we can prove that this lower bound is tight up to a $\log\log|S|$ multiplicative factor, by easily extending to the case of the LZ77-dictionary (which is dynamic), a result proved in [22] for static dictionaries. Precisely, it holds that $\frac{|\mathtt{LZ}_{f,g}(S)|}{|\mathsf{OPT}_{f,g}(S)|} \leq \frac{|f(|S|)|+|g(|S|)|}{|f(0)|+|g(0)|}$, which is upper bounded by $O(\log|S|)$ because $|f(|S|)| = |g(|S|)| = \Theta(\log|S|)$ and $|f(0)| = |g(0)| = O(1)$. To see this, let us assume that $\mathtt{LZ}_{f,g}(S)$ and $\mathsf{OPT}_{f,g}(S)$ are formed by ℓ_{lz} and ℓ_{opt} phrases respectively. Of course, $\ell_{lz} \leq \ell_{opt}$ because the greedy parsing is optimal with respect to the number of parsed phrases for S, whereas the other parser is optimal with respect to the number of bits in output. We then assume the worst-case scenario in which every phrase is encoded by $\mathtt{LZ}_{f,g}(S)$ with the longest encoding (namely, $f(|S|)|$ and $|g(|S|)|$ bits each) while $\mathsf{OPT}_{f,g}(S)$ uses the shortest one (namely, $|f(0)|$ and $|g(0)|$ bits each). Therefore, we have $\frac{|\mathtt{LZ}_{f,g}(S)|}{|\mathsf{OPT}_{f,g}(S)|} \leq \frac{\ell_{lz}(|f(|S|)|+|g(|S|)|)}{\ell_{opt}(|f(0)|+|g(0)|)} \leq \frac{|f(|S|)|+|g(|S|)|}{|f(0)|+|g(0)|} = \Theta(\log|S|)$.

[6] 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.

5. On Bit-Optimal Parsings and Shortest-Path problems. In this section we will describe how to compute efficiently the parsing that minimizes the final compressed size of S with respect to all possible LZ77-parsings. 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 $\langle i - p, j - i\rangle$ in case (2). The second case corresponds to copying a phrase longer than one single character[7].

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 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 integer 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 $\mathsf{OPT}_{f,g}(S)$ 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. $S = a^n$) thus resulting inefficient and actually unfeasible 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, provided that they satisfy Property 1 (see Section 2), and is $O(n \log n)$ for most known integer-encoding functions used in practice. 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.

5.1. A useful and 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 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 $\mathcal{G}(S)$. We can prove that the indices of the vertices in $FS(v)$ and $BS(v)$ form a *contiguous* range:

FACT 1. *Given a vertex v_i and let v_{i+x} and v_{i-y} be respectively the vertex with greatest index in $FS(v_i)$ and the smallest index in $BS(v_i)$, it holds*

- $FS(v_i) = \{v_{i+1} \ldots, v_{i+x-1}, v_{i+x}\}$ *and*
- $BS(v_i) = \{v_{i-y} \ldots, v_{i-2}, v_{i-1}\}$.

Furthermore, x and 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 derives from the definition of (v_i, v_{i+x}) which

[7]Notice that there may be several different candidate positions p from which we can copy the substring $S[i : j - 1]$. We can arbitrarily choose any position among the ones whose distance from i is encodable with the smallest number of bits (namely, $|f(d_{i,j})|$ bits is minimized).

represented a repeated substring in S. A similar argument holds for $BS(v_i)$. □

This means that if an edge (v_i, v_j) does exist in $\mathcal{G}(S)$, then there exist also all edges which are *nested* within it and are incident into one of its extremes, namely either v_i or v_j. 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. the shorter is the edge), the smaller is the cost of encoding the phrase $S[i : j-1]$.[8]

FACT 2. *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 observe that $S[i : j'-1]$ is a prefix of $S[i : j''-1]$ and thus the first substring occurs wherever the latter occurs. Therefore, we can always copy $S[i : j'-1]$ from the same position at which we copy $S[i : j''-1]$. By the Increasing Cost Property satisfied by f and g, we have that $|f(d_{i,j'})| \leq f|(d_{i,j''})|$ and $|g(\ell_{i,j'})| \leq |g(\ell_{i,j''})|$. □

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

DEFINITION 5.1. *An edge $(v_i, v_j) \in E$ is called*

- *$d-$*maximal *iff the next edge from v_i takes more bits to encode its distance: $|f(d_{i,j})| < |f(d_{i,j+1})|$;*
- *$\ell-$*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 ℓ-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 5.2. *There are at most $Q(f, n) + Q(g, n)$ maximal edges outgoing from any vertex v_i.*

Proof. By Fact 1, vertices in $FS(v_i)$ have indices in a range R, and by Fact 2, $c(v_i, v_j)$ 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. □

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 5.3. *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 π which traverses maximal edges only. Assume w.l.o.g. that π 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 $j > i_{r+1}$. Now, since $\mathcal{G}(S)$ is a DAG and indices in π are increasing, it must exist an index $i_h \geq i_{r+1}$ such that the index of that maximal edge j lies

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

in $[i_h, i_{h+1}]$. Since $(v_{i_h}, v_{i_{h+1}})$ is an edge of π and thus of the graph $\mathcal{G}(S)$, it does exist the edge $(v_j, v_{i_{h+1}})$ (by Fact 1), and by Fact 2 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. □

Theorem 5.3 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 the subgraph $\widetilde{\mathcal{G}}(S)$ consists of $n+1$ vertices and at most $n(Q(f,n)+Q(g,n))$ edges[9]. For Elias' codes [12], Fibonacci's codes [13], Rice's codes [30], 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, 21]. For the encoders used in `gzip`, it is $Q(f,n) = Q(g,n) = O(1)$ and thus, in these practical settings, we have $|\widetilde{\mathcal{G}}(S)| = O(n)$.

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 [9, Section 24.2]), here applied to work on the subgraph $\widetilde{\mathcal{G}}(S)$. Therefore, its correctness follows directly from Theorem 24.5 of [9] and our Theorem 5.3. However, it is clear that we cannot generate $\widetilde{\mathcal{G}}(S)$ by pruning $\mathcal{G}(S)$, so the key difficulty in computing the SSSP is 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 show that FSG takes $O(1)$ amortized time per edge and $O(n)$ space in total (although the size of $\mathcal{G}(S)$ may be $\omega(n)$). Combining this result with Lemma 5.2, we will obtain the following theorem:

THEOREM 5.4. *Given a string $S[1,n]$ drawn from an alphabet $\Sigma = [\sigma]$, and two integer-encoding functions f and g that satisfy Property 1, there exists a compressor that computes the (f,g)-optimal parsing of S based on a* LZ77*-dictionary by taking $O(n(Q(f,n)+Q(g,n)))$ time and $O(n)$ space in the worst case.*

Most of the integer-encoding functions used in practice are such that $Q(e,n) = O(\log n)$ [13]. This holds also for the following codes: Elias Gamma code, Elias Delta code, Elias Omega code, Fibonacci code, Variable-Byte code, Even-Rodeh codes, Nibbles code Rice codes or Boldi-Vigna Zeta codes [36, 13, 5]. Thus, by Theorem 5.4 we derive the following corollary that instantiates our result for many of such integers-encoding functions.

COROLLARY 5.5. *Given a string $S[1,n]$ drawn from an alphabet $\Sigma = [\sigma]$, and let f and g be chosen among the class of integer codes indicated above, the (f,g)-optimal parsing of S based on a* LZ77*-dictionary can be computed in $O(n \log n)$ time and $O(n)$ working space in the worst case.*

To the best of our knowledge, this result is the first one that answers positively to the question posed by Rajpoot and Sahinalp in [29, pag. 159]. The rest of this section will be devoted to prove Theorem 5.4, which is indeed the main contribution of this paper.

From Lemma 5.2 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. 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

[9] Observe that $|FS(v)| \leq Q(f,n) + Q(g,n)$, for any vertex v in $\widetilde{\mathcal{G}}(S)$ (Lemma 5.2).

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 5.1 and Fact 2, any edge following (v_i, v_{h_k}) in $FS(v_i)$ denotes a longer substring which must lie in a farther 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 ℓ-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 ℓ-maximal edge, because it needs some algebraic calculations, and the corresponding copied substring can then be inferred as a prefix of $S[i : h_k - 1]$.

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 [18] 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 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 kth pass, we logically partition the vertices of $\widetilde{\mathcal{G}}(S)$ in blocks of $|I_k|$ contiguous vertices, say $v_i, v_{i+1}, \ldots, v_{i+|I_k|-1}$, and compute all d-maximal edges which spread out from that block of vertices 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 our bit-optimal parser, and discarded as soon as the first vertex of the next block, i.e. $v_{i+|I_k|}$, needs to be processed. The next block of $|I_k|$ vertices is then fetched and the process repeats. All passes are executed in *parallel* over the I_k in order to guarantee that all d-maximal edges of v_i are available when processing this vertex. This means that, at any time, we have available d-maximal edges for all $Q(f, n)$ distinct blocks of vertices, one block for each interval I_k. Thus, the space occupancy is $\sum_{k=1}^{Q(f,n)} |I_k| = O(n)$. Observe that there exist other choices for the size of the blocks that allow to retain $O(n)$ working space. However, our choice has the additional advantage of making it possible an efficient computation of the d-maximal edges of vertices within each block.
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 $\sum_{k=1}^{Q(f,n)} (n/|I_k|)O(|I_k|) = O(n\,Q(f, n))$, and thus we will go to pay $O(1)$ amortized time per d-maximal edge. Combining this fact with the previous observation on the computation of the ℓ-maximal edges, we get Theorem 5.4 above.

Consider the kth 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* (of indices) $W_h = [h - r, h - l]$. 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

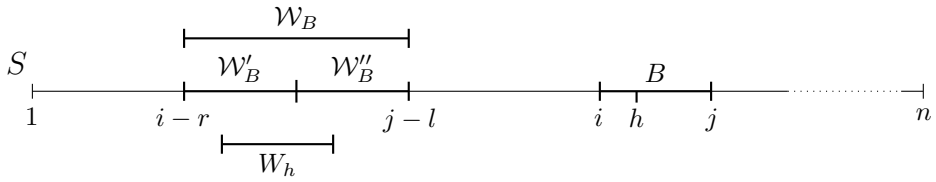

FIG. 5.1. *Interval $B = [i, j]$ with $j = i + |I_k| - 1$, window $\mathcal{W}_B$ and its two halves $\mathcal{W}'_B = W_i$ and $\mathcal{W}''_B = W_j$. It is also shown the window W_h of position $h \in B$.*

window $\mathcal{W}_B = W_i \cup W_{i+|I_k|-1}$ which merges the first and last window of positions that can be the (copy-)reference of any d-maximal edge outgoing from B. Note that $|\mathcal{W}_B| = 2|I_k| - 1$ (see Figure 5.1) and it spans all positions where the copy of a d-maximal edge outgoing from a vertex in B can occur.

The next fact is crucial to fast compute all these d-maximal edges via an indexing data structures built over S:

FACT 3. *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 longest common prefix (shortly, lcp) with S_h which is the maximum lcp between S_h and all other suffixes in W_h.*

Proof. Let the edge (v_h, v_{h+q+1}) be a d-maximal edge of d-cost $c(I_k)$. The reference copy of $S[h, h+q]$ must start in W_h, having distance in I_k. Among all positions s in W_h take one whose suffix S_s shares the lcp with S_h, clearly $q \leq \mathsf{lcp}$. We have to show that $q = \mathsf{lcp}$, so this is maximal. Of course, there may exist many such positions, we take just one of them.

By definition of d-maximality, any other edge longer edge $(v_h, v_{h+q''})$, with $q'' > q$, represents a substring whose reference-copy occurs in a farther window before W_h in S. So any other position $s' \in W_h$ must be the starting position of a reference-copy that is shorter than q. □

Notice that a vertex v_h may not have a d-maximal edge of d-cost $c(I_k)$. Indeed, suffix S_s may share the maximum lcp with suffix S_h in the window W_h, but a longer lcp can be shared with a suffix in a window closer to S_h. Thus, from v_h we can reach a farther vertex at a lower cost implying that v_h has no d-maximal edge of cost $c(I_k)$. Hereafter we call the position s of Fact 3 *maximal position* for vertex v_h whenever it induces a d-maximal edge for v_h.

Our algorithm will compute the maximal positions of every vertex v_h in B and every cost $c(I_k)$. If no maximal position does exist, v_h will be assigned an *arbitrary* position. 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 5.2 and can be created efficiently in $O(|I_k|)$ time and space, for each block of distances I_k, as we required above.

Fact 3 relates 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 to some string-matching data structure, like the compact trie $\mathcal{T}_B$, built over the suffixes of S which start in the range of positions $B \cup \mathcal{W}_B$. Compacted trie $\mathcal{T}_B$ takes $O(|B| + |\mathcal{W}_B|) = O(|I_k|)$ space (namely, proportional to the number of indexed strings), 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. The proof of this result may be of independent interest, and it is deferred to Subsection 5.3. The key idea is to

exploit the fact that the algorithm deploying this trie (and that we detail below) does not make any assumptions on the edge-ordering of $\mathcal{T}_B$, because it just computes (sort of) lca-queries on its structure.

So, let us assume that we are given the trie $\mathcal{T}_B$. We notice that maximal position s for a vertex v_h in B having d-cost $c(I_k)$ can be computed by *finding a leaf of $\mathcal{T}_B$ which is labeled with an index s that belongs to the range W_h and shares the maximum* lcp *with the suffix spelled out by the leaf labeled h.*[10] This actually corresponds to finding a leaf whose label s belongs to the range W_h and has the deepest 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, one could think to resort to proper predecessor/successor queries on a suitable dynamic set of suffixes in W_h. The idea is to consider the suffixes of W_h and to identify the predecessor and the successor of h in the lexicographically order. The answer is the one among these two suffixes that shares the longest common prefix with S_h. Unfortunately, this would take $\omega(1)$ time per query 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 5.6). 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 for each node u of $\mathcal{T}_B$. For each node u, we denote by $a(u)$ the smallest position belonging to B and labeling a leaf in the subtree rooted at u. Value of $a(u)$ is undefined ($\perp$) whenever such a smallest position does not exist. Computing the value $a()$ for all nodes u in $\mathcal{T}_B$ takes $O(|\mathcal{T}_B|) = O(|I_k|)$ time and space via a traversal of the trie $\mathcal{T}_B$. By discussion above, it follows that, for each node u, vertex $v_{a(u)}$ is exactly the solely vertex for which we have to identify its maximal position.

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 5.1). 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 positions 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

[10] Observe that there may be several leaves having these characteristics. We can arbitrarily choose one of them because they denote copies of the same phrase that can be encoded with the same number of bits for the length and for the distance (i.e., $c(I_k)$ bits).

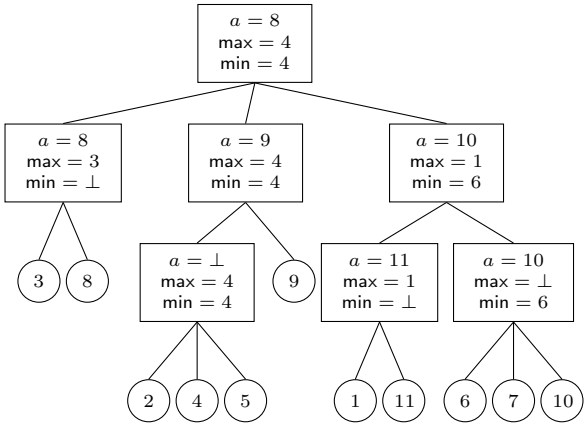

FIG. 5.2. *The picture shows a running example for a(n imaginary) trie $\mathcal{T}_B$ and our algorithm. We are assuming that $I_k = [4, 7]$ and $B = [8, 11]$. Thus, $|I_k| = 4$, $\mathcal{W}'_B = [1, 4]$ and $\mathcal{W}''_B = [4, 7]$. For each node of the tree, we report the values of $a()$, $\mathsf{max}()$ and $\mathsf{min}()$.*

$\mathsf{max}(u)$ and $\mathsf{min}(u)$. This computation takes $O(|I_k|)$ time with a post-order traversal of $\mathcal{T}_B$. We can now efficiently compute $\mathsf{mp}[h]$ as a maximal position for v_h, if it exists, or otherwise set $\mathsf{mp}[h]$ arbitrarily. We initially set all 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 $\mathsf{mp}[a(u)] = \mathsf{nil}$: If $\mathsf{min}(u) \in W_{a(u)}$, we set $\mathsf{mp}[a(u)] = \mathsf{min}(u)$; if $\mathsf{max}(u) \in W_{a(u)}$, we set $\mathsf{mp}[a(u)] = \mathsf{max}(u)$.[11] We finally check if $\mathsf{mp}[a(u)]$ is still nil and we set $\mathsf{mp}[a(u)] = a(\mathsf{parent}(u))$ whenever $a(u) \neq a(\mathsf{parent}(u))$. This last check is needed (see proof of Lemma 5.6) to manage the case in which we can copy the phrase starting at position $a(u)$ from position $a(\mathsf{parent}(u))$ 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 5.4 follows.

Figure 5.2 shows a running example of our algorithm in which $|I_k| = 4$, $B = [8, 11]$ and the trie $\mathcal{T}_B$ is the one shown (it is an imaginary trie, just for illustrative purposes). It is $\mathcal{W}'_B = [1, 4]$ and $\mathcal{W}''_B = [4, 7]$. Notice that some $a(u)$-values are $\perp$ because the leaves descending from that node are labeled with positions which lie outside B; otherwise $a(u)$ is the smallest position descending from u and belonging to B. Now take some node u, say the rightmost one at the last level of the trie, it has $a(u) = 10$ and its reference window $W_{10} = [10 - 7, 10 - 4] = [3, 6]$ for which the copies have cost $c(I_k)$. We can compute $\mathsf{min}(u) = 6$ because it is the minimum of the leaves descending from u and whose value belongs to $W''_B = [4, 7]$. Instead, we have $\mathsf{max}(u) = \perp$ because no leaf-value belongs $W'_B = [1, 4]$. In this case, $\mathsf{mp}[10]$ is correctly set to 6 which is the maximal position for v_{10}. The other results of our algorithm are $\mathsf{mp}[8] = 3$, $\mathsf{mp}[9] = 4$ and $\mathsf{mp}[11] = 10$.[12] Observe that 3 and 4 are indeed maximal positions for, respectively, v_8 and v_9. Moreover, we notice that v_9 has three possible maximal positions (namely, 2, 4 and 5). Choosing arbitrarily one of them does not change neither the length of the copied substring nor the cost of its encoding. We finally notice that v_{11} has no maximal position with cost $c(I_k)$, in fact values of min and max on the node with $a = 11$ are not in W_{11}. Observe that we can indeed copy the same

[11] The value of $\mathsf{mp}[a(u)]$ can be arbitrarily set to $\mathsf{min}(u)$ or $\mathsf{max}(u)$ whenever both $\mathsf{min}(u)$ and $\mathsf{max}(u)$ belong to $W_{a(u)}$.

[12] Observe that we obtain $\mathsf{mp}[11] = 10$ by setting $\mathsf{mp}[a(u)] = a(\mathsf{parent}(u))$ at the node with $a = 11$.

substring from positions 7 and 10, the latter copy is less expensive since it is closer to position 11.

LEMMA 5.6. *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* $\mathsf{mp}[h]$ *is equal to its maximal position.*

Proof. Take $B = [i, i + |I_k| - 1]$ 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 $\mathsf{parent}(u_z)$ does exist, then it is $a(\mathsf{parent}(u_z)) < h$. There are two cases for the final value stored in $\mathsf{mp}[h]$.

Case 1. Suppose that $\mathsf{mp}[h] \in W_h$. We want to prove that $\mathsf{mp}[h]$ is the index of the leaf which has the deepest lca with h among all the other leaves labeled with an index in W_h (hence it has maximal length). Let $u_x \in \pi$ be the node in which the value of $\mathsf{mp}[h]$ is assigned, so $a(u_x) = h$ and the ancestors of u_x on π will not change $\mathsf{mp}[h]$ because the algorithm will find $\mathsf{mp}[h] \neq \mathsf{nil}$. Assume now that there exists at least another position in W_h whose leaf has a deeper lca with leaf h. This 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 $\mathsf{max}(u_l)$ or $\mathsf{min}(u_l)$ must lie in W_h and thus the algorithm has set $\mathsf{mp}[h]$ to one of these positions, because of the post-order visit of $\mathcal{T}_B$ and the check on $\mathsf{mp}[a(u_l)] = \mathsf{nil}$. Therefore $\mathsf{mp}[h]$ must be the index of the leaf having the deepest lca with h, and thus by Fact 3 it is its maximal position.

Case 2. Suppose that $\mathsf{mp}[h] \notin W_h$ 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 lcp with S_h, and let l be the lcp-length. Values $\mathsf{min}(u_i)$ and $\mathsf{max}(u_i)$ do not belong to W_h, for any node $u_i \in \pi$ with $a(u_i) = h$, otherwise $\mathsf{mp}[h]$ would have been assigned with an index in W_h (contradicting the hypothesis). Thus the value of $\mathsf{mp}[h]$ remains nil up to node u_z where it is set to $a(\mathsf{parent}(u_z))$. 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 $\mathsf{parent}(u_z)$ to the root of $\mathcal{T}_B$, and, as a result, it is $\mathsf{lcp}(S_{a(\mathsf{parent}(u_z))}, S_h) \geq \mathsf{lcp}(S_s, S_h) = l$. By observing that $a(\mathsf{parent}(u_z)) < a(u_z) = h$, $a(\mathsf{parent}(u_z))$ belongs to B (both by definition of a-value), and $\mathsf{mp}[h] = a(\mathsf{parent}(u_z)) \notin W_h$ (by assumption), it follows that we can copy from position $a(\mathsf{parent}(u_z))$ at a cost smaller than $c(I_k)$ a substring longer than the one we can copy from s. So we found an edge from v_h with smaller d-cost and at least the same length. This way v_h has no d-maximal edge of cost $c(I_k)$ in $\widetilde{\mathcal{G}}(S)$. □

5.3. On the optimal construction of $\mathcal{T}_B$. In the discussion above we left out the explanation on 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 $|I_k|$ indexed suffixes and the alphabet size σ. To achieve this result we deploy the crucial fact that the algorithm of the previous section does not make any assumption on the ordering of the children of $\mathcal{T}_B$'s nodes, because it just computes (sort of) lca-queries on its structure.

First of all, 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]). This takes $O(n)$ time and space.

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 σ. Then, we define $\hat{S}$ as the string S which has been transformed by re-mapping symbols according to table T (namely, those occurring in $S[i, i+3|I_k|-1]$). We can prove the following Lemma.

LEMMA 5.7. *Let $S_i, \ldots, S_j$ be a contiguous sequence of suffixes in S. The re-mapped suffixes $\hat{S}_i \ldots \hat{S}_j$ can be lexicographically sorted in $O(j-i+1)$ time.*

Proof. Consider the string of pairs $w = \langle \hat{S}[i], b_i \rangle \ldots \langle \hat{S}[j], b_j \rangle \$$, where b_h is 1 if $\hat{S}_{h+1} > \hat{S}_{j+1}$, -1 if $\hat{S}_{h+1} < \hat{S}_{j+1}$, 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 $\hat{S}_{p+i} > \hat{S}_{q+i}$ iff $w_p > w_q$ and thus comparing the corresponding suffixes of w. In fact, suppose that $w_p > w_q$ and set $r = \mathsf{lcp}(w_p, w_q)$. We have that $w[p+r] = \langle \hat{S}[p+i+r], b_{p+i+r} \rangle > \langle \hat{S}[q+i+r], b_{q+i+r} \rangle = w[q+r]$. Hence we have that either $\hat{S}[p+i+r] > \hat{S}[q+i+r]$ or $b_{p+i+r} > b_{q+i+r}$, which means $b_{p+i+r} = 1$ and $b_{q+i+r} = 0$. The latter actually means that $\hat{S}_{p+i+r+1} > \hat{S}_{j+1} \geq \hat{S}_{q+i+r+1}$. In any case, it follows that $\hat{S}_{p+i+r} > \hat{S}_{q+i+r}$ and thus $\hat{S}_{p+i} > \hat{S}_{q+i}$, since their first r symbols are equal.

This implies that sorting the suffixes $\hat{S}_i, \ldots, \hat{S}_j$ 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 $\hat{S}_z$ with $\hat{S}_{j+1}$ takes $O(1)$ time via an lcp-query on S (using the proper data structure above) and a check at their first mismatch. □

Lemma 5.7 allows us to generate the compact trie of $\hat{S}_i, \ldots, \hat{S}_{i+3|I_k|-1}$, which is equal to the (unordered) compacted trie of $S_i, \ldots, S_{i+3|I_k|-1}$ 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.

6. An experimental support to our theoretical findings. In this section we provide an experimental support to our findings of Section 4, and compare our proposals of Sections 3 and 5 with some state-of-the-art compressors over few freely available text collections. Table 6 reports our experimental results.

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. This motivates our main study for $M = n$, even if our results extend to the bounded-window case too.

Then consider `Rightmost-LZ77` (Section 3), which uses an unbounded window

Compressor	english [17]	C/C++/Java src [17]
gzip -9	37.52%	23.29%
bzip2 -9	28.40%	19.78%
boosterOpt	20.62%	17.36%
lzma2 -9	19.95%	16.06%
Fixed-LZ77	26.19%	24.63%
Rightmost-LZ77	25.02%	21.21%
BitOptimal-LZ77	22.11%	18.97%
Compressor	HTML [4]	Avg Dec. time (sec)
gzip -9	11.99%	0.7
bzip2 -9	8.10%	6.3
boosterOpt	4.45%	20.2
lzma2 -9	4.62%	2.1
Fixed-LZ77	6.69%	0.8
Rightmost-LZ77	6.16%	0.9
BitOptimal-LZ77	5.68%	0.9

TABLE 6.1

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.*

and selects the rightmost copy of the currently longest phrase. As expected, this parsing combined with the use of variable-length integer encoders improves Fixed-LZ77, thus sustaining in practice the starting point of our theoretical investigation.

Finally, we tested our bit-optimal compressor BitOptimal-LZ77[13] finding that it improves Rightmost-LZ77, as theoretically predicted in Lemma 4.1. Surprisingly, BitOptimal-LZ77 significantly improves bzip2 (which uses a bounded window) and comes close to the *booster* tool (which uses an unbounded window [16]) and lzma2[14] (which is an LZ77-based compressor using a sophisticated encoding algorithm). 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 decompressing bzip2's or booster's compressed files, and three times faster than lzma2. 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).

As far as construction time is concerned, BitOptimal-LZ77 is slower than other LZ77-based compressors by factors that range from 2 (w.r.t. lzma2) to 20 (w.r.t. gzip). Our preliminary implementation does not follow the computation of maximal edges as described in Subsection 5.2. Instead, we use the less efficient solution mentioned in the previous section which resorts to binary balanced search trees to solve predecessor/successor queries on a suitable dynamic set of suffixes in W_h. This approach gives an easier solution to implement which, however, loses a factor $\log n$ in time. We believe that a more engineered implementation could sensibly reduce the construction time of BitOptimal-LZ77. This what we plan to investigate in the near

[13] Algorithms Rightmost-LZ77 and BitOptimal-LZ77 encode copy-distances and lengths by using a variant of Rice codes in which we have not just one bucketing of size 2^k, rather we have a series of buckets of increasing size, fixed in advance.

[14] Lzma2 home page http://7-zip.org/.

future.

7. Conclusion. In this paper we have addressed the question of bit-optimality in LZ77-parsing by investigating some theoretical questions whose results have been checked experimentally on some test data, showing that LZ-approaches are still promising. An algorithmic-engineering effort is now needed to tune the performance of the proposed bit-optimal algorithms, by possibly designing integer encoders which are suited for the data collections in input.

As far as theoretical issues are concerned, the main result of this paper shows that the bit-optimal LZ77-parsing of a string $S[1,n]$ can be computed in $O(n \log n)$ time and $O(n)$ optimal working-space in the worst case, for most integer-encoding functions f, g (see Corollary 5.5). To the best of our knowledge, this is the first result that answers positively to the question posed by Rajpoot and Sahinalp in [29, pag. 159].

Our bit-optimal parsing scheme can be easily 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], we have $Q(f, M) = O(1)$, and thus the running time of our algorithm is $O(n\,Q(g, n))$. This complexity could be further refined as $O(n\,Q(g, \ell))$ by considering the length ℓ of the *longest repeated substring* in S. 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 time complexity of our algorithm results $O(n \log \log n)$ for this class of strings.

We leave two main open questions. The first one refers to the paradigm mentioned above, namely "compress once & decompress many times" which is typical of Web search engines and IR storage systems, e.g., BigTable platform of Google or Hadoop platform of Yahoo!. The decompression speed is crucial in these settings, and indeed it is not difficult to come up with examples of LZ-parsings which are succinctly compressible but induce many random I/Os (or cache faults) in the decoding phase, as well as find LZ-parsings which are $(1+\epsilon)$-close to the optimality in space but much more I/O-efficient to decompress. To overcome these limitations, some practical solutions trade space occupancy for decompression efficiency by hand-tuning several parameters which significantly impact on the final result and depend on the underlying data type to be compressed! An example is the Snappy-compressor adopted by Google in BigTable. On the other hand, the theory literature offers solutions which are able to optimize only one between decompression time and compressed space (such as our proposal in this paper). Therefore, we foresee the design of a space/time controlled (de)compression tool which addresses the following problem [14]: Given two approximation factors ϵ, δ, can we design a compression format that is decodable in $O((1+\delta)T_{opt})$ I/Os and takes $(1+\epsilon)S_{opt}$ space? Where T_{opt} is the optimal number of I/Os required to decompress (part of) the compressed data, and S_{opt} is the optimal space in which that data can be compressed. More ambitiously we could aim at fixing either ϵ or δ equal to zero, and thus optimize one of the two resources given an upper-bound on the other. We may argue favorably about the solvability of this question because it has been already addressed with success in the simpler context of Dictionary Compression by [3] with the so called "Locality Preserving Front-Coding".

The second open question asks to extend our results to *statistical* encoding functions like Huffman or Arithmetic coders applied on the integral range $1 \ldots n$ [36]. They do not necessarily satisfy the increasing cost Property 1 because it might be the

case that $x < y$ but $|f(x)| > |f(y)|$, given that 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. *Information Processing Letters*, 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 (STOC)*, pages 295–304, 1999.

[3] M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul. Cache-oblivious string b-trees. In *PRocs of the 25th Symposium on Principles of Database Systems (PODS)*, pages 233–242. ACM, 2006.

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

[5] P. Boldi and S. Vigna. Codes for the world wide web. *Internet Mathematics*, 2(4), 2005.

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

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

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

[9] 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.

[10] G. Cormode and S. Muthukrishnan. Substring compression problems. In *16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 321–330, 2005.

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

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

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

[14] P. Ferragina. Data structures: Time, I/Os, entropy, joules! In *Procs of the 18th European Symposium on Algorithms (ESA)*, volume 6347 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2010.

[15] P. Ferragina, R. Giancarlo, and G. Manzini. The engineering of a compression boosting library: Theory vs practice in bwt compression. In *14th annual European symposium on Algorithms (ESA)*, pages 756–767, 2006.

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

[17] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics*, 13, 2008.

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

[19] T. Gagie and G. Manzini. Space-conscious compression. In *Mathematical Foundations of Computer Science (MFCS)*, pages 206–217, 2007.

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

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

[22] 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.

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

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

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

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

[27] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 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, 4th Edition*. Springer Verlag, 2006.

[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. G. 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 Journal on Computing*, 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. Classification with finite memory revisited. *IEEE Transactions on Information Theory*, 53(12):4413–4421, 2007.

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

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