Algorithmica (2011) 61:51–74
DOI 10.1007/s00453-010-9437-6

On Optimally Partitioning a Text to Improve Its Compression

Paolo Ferragina · Igor Nitto · Rossano Venturini

Received: 22 February 2010 / Accepted: 19 July 2010 / Published online: 7 August 2010
© Springer Science+Business Media, LLC 2010

Abstract In this paper we investigate the problem of partitioning an input string T in such a way that compressing individually its parts via a base-compressor C gets a compressed output that is shorter than applying C over the entire T at once. This problem was introduced in Buchsbaum et al. (Proc. of 11th ACM-SIAM Symposium on Discrete Algorithms, pp. 175–184, 2000; J. ACM 50(6):825–851, 2003) in the context of table compression, and then further elaborated and extended to strings and trees by Ferragina et al. (J. ACM 52:688–713, 2005; Proc. of 46th IEEE Symposium on Foundations of Computer Science, pp. 184–193, 2005) and Mäkinen and Navarro (Proc. of 14th Symposium on String Processing and Information Retrieval, pp. 229–241, 2007). Unfortunately, the literature offers poor solutions: namely, we know either a cubic-time algorithm for computing the optimal partition based on dynamic programming (Buchsbaum et al. in J. ACM 50(6):825–851, 2003; Giancarlo and Sciortino in Proc. of 14th Symposium on Combinatorial Pattern Matching, pp. 129–143, 2003), or few heuristics that do not guarantee any bounds on the efficacy of their computed partition (Buchsbaum et al. in Proc. of 11th ACM-SIAM Symposium on Discrete Algorithms, pp. 175–184, 2000; J. ACM 50(6):825–851, 2003), or algorithms that are efficient but work in some specific scenarios (such as the Burrows-Wheeler Transform, see e.g. Ferragina et al. in J. ACM 52:688–713, 2005; Mäkinen and Navarro in Proc. of 14th Symposium on String Processing and

The first author has been supported in part by a Yahoo! Research grant, by the MIUR-PRIN project "Mad Web", and by the MIUR-FIRB project "Linguistica 2006".

P. Ferragina (✉) · I. Nitto
Dipartimento di Informatica, University of Pisa, Largo Bruno Pontecorvo 3, 56127 Pisa, Italy
e-mail: ferragina@di.unipi.it

I. Nitto
e-mail: nitto@di.unipi.it

R. Venturini
ISTI-CNR, Via G. Moruzzi 1, 56124 Pisa, Italy
e-mail: rossano.venturini@isti.cnr.it

Information Retrieval, pp. 229–241, 2007) and achieve compression performance that might be worse than the optimal-partitioning by a $\Omega(\log n / \log\log n)$ factor. Therefore, computing efficiently the optimal solution is still open (Buchsbaum and Giancarlo in Encyclopedia of Algorithms, pp. 939–942, 2008). In this paper we provide the first algorithm which computes in $O(n \log_{1+\varepsilon} n)$ time and $O(n)$ space, a partition of T whose compressed output is guaranteed to be no more than $(1+\varepsilon)$-worse the optimal one, where ε may be any positive constant fixed in advance. This result holds for any base-compressor C whose compression performance can be bounded in terms of the zero-th or the k-th order empirical entropy of the text T. We will also discuss extensions of our results to `BWT`-based compressors and to the compression booster of Ferragina et al. (J. ACM 52:688–713, 2005).

Keywords Data compression · Dynamic programming · Compression boosting · Table compression · Empirical entropy · Burrows-Wheeler transform · Arithmetic and Huffman coding

1 Introduction

Reorganizing data in order to improve the performance of a given compressor C is a recent and important paradigm in data compression (see e.g. [5, 12]). The basic idea consist of *permuting* the input string T to form a new string T' which is then *partitioned* into substrings $T' = T'_1 T'_2 \dots T'_h$ that are finally compressed *individually* by the base compressor C. The goal is to find the best instantiation of the two steps Permuting + Partitioning so that the compression of the individual substrings T'_i minimizes the total length of the compressed output. This approach (hereafter abbreviated as PPC) is clearly *at least* as powerful as the classic data compression approach that applies C to the entire T: just take the identity permutation and set $h = 1$. The question is whether it can be *more powerful* than that!

Intuition leads to think favorably about it: by grouping together objects that are "related", one can hope to obtain better compression even using a very weak compressor C. Surprisingly enough, this intuition has been sustained by convincing theoretical and experimental results only recently. These results have investigated the PPC-paradigm under various angles by considering: different data formats (strings [12], trees [13], tables [5], etc.), different granularities for the items of T to be permuted (chars, node labels, columns, blocks [1, 24], files [7, 30], etc.), different permutations (see e.g. [7, 19, 31, 32]), different base compressors to be boosted (0-th order compressors, `gzip`, `bzip2`, etc.). Among these plethora of proposals, we survey below the most notable examples which are useful to introduce the problem we attack in this paper, and refer the reader to the cited bibliography for other interesting results.

The PPC-paradigm was introduced in [4], and further elaborated upon in [5]. In these papers T is a *table* formed by fixed size columns, and the goal is to permute the columns in such a way that individually compressing contiguous groups of them gives the shortest compressed output. The authors of [5] showed that the PPC-problem in its full generality is MAX-SNP hard, devised a link between PPC and the classical asymmetric TSP problem, and then resorted known *heuristics* to find approximate solutions based on several measures of correlations between the table's columns. For the

grouping they proposed either an optimal but very slow approach, based on Dynamic Programming (shortly DP), or some very simple and fast algorithms which however did not have any guaranteed compression bounds. Nonetheless, experiments showed that these heuristics achieve significant improvements over the classic `gzip`, when it is applied on the serialized original T (row- or column-wise). Also, they showed that the combination of the TSP-heuristic with the DP-optimal partitioning is better, but it is too slow to be used in practice even on short files because of the DP-cubic time complexity.[1]

When T is a text string, the most famous instantiation of the PPC-paradigm has been obtained by combining the Burrows and Wheeler Transform [6] (shortly `BWT`) with a context-based grouping of the input symbols, which are finally compressed via proper zero-th order-entropy compressors (like `MTF`, `RLE`, Huffman, Arithmetic, or their combinations, see e.g. [33]). Here the PPC-paradigm takes the name of *compression booster* [12] because the net result it produces is to boost the performance of the base compressor C from zero-th order-entropy bounds to k-th order entropy bounds, simultaneously over all $k \geq 0$. In this scenario the permutation acts on single symbols, and the partitioning/permuting steps deploy the context (substring) following each symbol in the original string in order to identify "related" symbols which must be therefore compressed together. Recently [19] investigated whether do exist other permutations of the symbols of T which admit effective compression and can be computed/inverted fast. Unfortunately they found a connection between table compression and the `BWT`, so that many natural similarity-functions between contexts turned out to induce MAX-SNP hard permuting problems! Interesting enough, the `BWT` seems to be the unique highly compressible permutation which is fast to be computed and achieves effective compression bounds. Several other papers have given an analytic account of this phenomenon [16, 22, 26, 27] and have shown, also experimentally [15], that the partitioning of the BW-transformed data is a key step for achieving effective compression ratios. Optimal partitioning is actually even more mandatory in the context of labeled-tree compression where a `BWT`-inspired transform, called `XBW`-transform in [13, 14], allows to produce permuted strings with a strong clustering effect. Starting from these premises, [18] attacked the computation of the optimal partitioning of T via a DP-approach, which turned to be very costly; then [12] (and subsequently many other authors, see e.g. [13, 16, 26]) proposed solutions which are not optimal but, nonetheless, achieve interesting k-th order-entropy bounds. This is indeed a subtle point which is frequently neglected when dealing with compression boosters, especially in practice, and for this reason we detail it more clearly in Sect. 8, in which we show an infinite class of strings for which the compression achieved by the classic booster is far from the optimal partitioning by a multiplicative factor $\Omega(\log n / \log\log n)$.

Finally, there is another scenario in which the computation of the optimal partition of an input string for compression boosting can be successful and occurs when T is a single (possibly long) file on which we wish to apply classic data compressors, such

[1] Page 836 of [5] says: "computing a good approximation to the TSP reordering before partitioning contributes significant compression improvement at minimal time cost. [...] This time is negligible compared to the time to compute the optimal, contiguous partition via Dynamic Programming."

as `gzip`, `bzip2`, `ppm`, etc. [33]. Note that how much redundancy can be detected and exploited by these compressors depends on their ability to "look back" at the previously seen data. However, such ability has a cost in terms of memory usage and running time, and thus most compression systems provide a facility that controls the amount of data that may be processed at once—usually called the *block size*. For example the classic tools `gzip` and `bzip2` have been designed to have a small memory footprint, up to few hundreds KBs. More recent and sophisticated compressors, like `ppm` [33] and the family of `BWT`-based compressors [15], have been designed to use block sizes of up to a few hundreds MBs. But using larger blocks to be compressed at once does not necessarily induce a better compression ratio! As an example, let us take C as the simple Huffman or Arithmetic coders and use them to compress the text $T = 0^{n/2}1^{n/2}$: There is a clear difference whether we compress individually the two halves of T (achieving an output size of about $O(\log n)$ bits) or we compress T as a whole (achieving $n + O(\log n)$ bits). The impact of the block size is even more significant as we use more powerful compressors, such as the k-th order entropy encoder `ppm` which compresses each symbol according to its preceding k-long context. In this case take the string $T = (2^k 0)^{\frac{n}{2(k+1)}}(2^k 1)^{\frac{n}{2(k+1)}}$ drawn from the ternary alphabet $\Sigma = 0, 1, 2$; observe that if we divide T in two halves and compress them individually via a k-order compressor, the output size is about $O(\log n)$ bits, but if we compress the entire T at once by that k-order compressor then the output size turns to be much longer, i.e. $\frac{n}{k+1} + O(\log n)$ bits. Therefore the choice of the block size has an impact that cannot be underestimated and, additionally, it is made even more problematic by the fact that it is not necessarily the same along the whole file we are compressing because it depends on the distribution of the repetitions within it. This problem is even more challenging when T is obtained by concatenating a collection of files via any permutation of them: think to the serialization induced by the Unix `tar` command, or other more sophisticated heuristics like the ones discussed in [7, 29–31]. In these cases, the partitioning step looks for *homogeneous* groups of contiguous files which can be effectively compressed together by the base-compressor C. More than before, taking the largest memory-footprint offered by C to group the files and compress them at once is not necessarily the best choice because real collections are typically formed by homogeneous groups of dramatically different sizes (e.g. think to a Web collection and its different kinds of pages, see e.g. [10]). Again, in all those cases we could apply the optimal DP-based partitioning approach of [5, 18], but this would take more than cubic time (in the overall input size $|T|$) thus resulting unusable even on few MBs of input data!

In summary the efficient computation of an optimal partitioning of the input text for compression boosting is an important and still open problem of data compression (see [3]). The goal of this paper is to make a step forward by providing the first efficient approximation algorithm for this problem, formally stated as follows.

Let C be the base compressor we wish to boost, and let $T[1, n]$ be the input string we wish to partition and then compress by C. So, we are assuming that T has been (possibly) permuted in advance, and we are concentrating only on the last two steps of the PPC-paradigm. Now, given a partition P of the input string into contiguous substrings, say $T = T_1 T_2 \ldots T_h$, we denote by $\texttt{Cost}(P)$ the cost of this partition and measure it as $\sum_{i=1}^{h} |C(T_i)|$, where $|C(\alpha)|$ is the length in bit of the string α

compressed by C. The problem of *optimally partitioning* T according to the base-compressor C consists then of computing the partition $P_{\texttt{opt}}$ that achieves the minimum cost, namely $P_{\texttt{opt}} = \min_P \texttt{Cost}(P)$, and thus the shortest compressed output.[2]

As we mentioned above $P_{\texttt{opt}}$ might be computed via a Dynamic-Programming approach [5, 18]. Define $E[j]$ as the cost of the optimum partitioning of $T[1, j-1]$, and set $E[1] = 0$. Then, for each $j > 1$, we can compute $E[j]$ as the $\min_{1 \le i < j} E[i] + |C(T[i, j-1])|$. At the end $E[n+1]$ gives the cost of $P_{\texttt{opt}}$, which can be explicitly determined by standard back-tracking over the DP-array E. Unfortunately, this solution requires to run C over $\Theta(n^2)$ substrings of average length $\Theta(n)$, for an overall $\Theta(n^3)$ time cost in the worst case which is clearly unfeasible even on small input sizes n.

In order to overcome this computational bottleneck we make two crucial observations: (1) instead of applying C over each substring of T, we use an entropy-based estimation of C's compressed output that can be computed efficiently and incrementally by suitable dynamic data structures; (2) we relax the requirement for an exact solution to the optimal partitioning problem, and aim at finding a partition whose cost is no more than $(1+\varepsilon)$ worse than $P_{\texttt{opt}}$, where ε may be any positive constant fixed in advance. Item (1) takes inspiration from the heuristics proposed in [4, 5], but it is executed in a more principled way because our entropy-based cost functions reflect the real behavior of modern compressors, and our dynamic data structures allow the efficient estimation of those costs without their re-computation from scratch at each substring (as instead occurred in [4, 5]). For item (2) it is convenient to resort to a well-known reduction from solutions of dynamic programming recurrences to Single Source Shortest path (SSSP) computation over weighted DAGs (see e.g. [8]). In our case, the solution for the optimal partitioning problem can be rephrased as a SSSP-computation over a weighted DAG consisting of n nodes and $O(n^2)$ edges whose costs are derived from item (1). By exploiting some interesting structural properties of this graph, we are able to restrict the computation of that SSSP to a subgraph consisting of $O(n \log_{1+\varepsilon} n)$ edges only. The technical part of this paper (see Sect. 4) will show that we can build this graph on-the-fly as the SSSP-computation proceeds over the DAG via the proper use of time-space efficient dynamic data structures. The final result will be to show that we can $(1+\varepsilon)$-approximate $P_{\texttt{opt}}$ in $O(n \log_{1+\varepsilon} n)$ time and $O(n)$ space, for both zero-th order compressors (like Huffman and Arithmetic [33], Sect. 5) and k-th order compressors (like ppm [33], Sect. 6). We will also extend these results to the class of BWT-based compressors, when T is a collection of texts, by introducing a poly-logarithmic slowdown (Sect. 7), as well as to the compression booster of [12] (Sect. 8).

We point out that the result on zero-th order compressors is interesting in its own from both the experimental side, since *Huffword* compressor is the standard choice for the storage of Web pages [33], and from the theoretical side since it can be applied to the compression booster of [12] to obtain a fast approximation of the optimal partition of $\texttt{BWT}(T)$ in $O(n \log_{1+\varepsilon} n)$ time. This latter result improves the algorithm of [12] both in time complexity, since that takes $O(n\sigma)$ time, and in compression ratio

[2] We are assuming that $C(\alpha)$ is a prefix-free encoding of α, so that we can concatenate the compressed output of many substrings and still be able to recover them via a sequential scan.

(for details see Sect. 8). The case of a large alphabet (namely, $\sigma = \Omega(\text{polylog}(n))$) is particularly interesting whenever we consider either a word-based BWT [28] or the XBW-transform over labeled trees [12]. Finally, we mention that our results apply also to the practical case in which the base compressor C has a maximum (block) size B of data it can process at once (see above the case of gzip, bzip2, etc.). In this situation the time performance of our solution is $O(n \log_{1+\varepsilon}(B \log \sigma))$.

The map of the paper is as follows. Section 2 introduces some basic notation and terminology. Section 3 presents a general technique to efficiently approximate the solutions of Dynamic Programming Recurrences provided that a special (but common) property holds for them. Here we first use the reduction from the problem of solving Dynamic Programming Recurrences to the problem of computing a SSSP over a proper weighted DAG. Then, we show how to prune that graph in order to significantly reduce the number of its edges in a way that the shortest path distances are *almost* preserved. We present this strategy in a general form since we believe that it could be used to speed up other algorithms based on the dynamic programming paradigm. In Sect. 4 we show how to use this technique to solve the optimal partitioning problem. The subsequent sections address the problem of incrementally and efficiently computing the edge costs of the weighted DAG as they are needed by the SSSP-computation, distinguishing the two cases of zero-th order estimators (Sect. 5) and k-th order estimators (Sect. 6), and the situation in which the base compressor C is a BWT-based compressor and T is a collection of files (Sect. 7). Finally, Sect. 8 focuses on the application of our methods to the compression-boosting technique of [12].

2 Background and Notations

2.1 Empirical Entropy and Compressive Estimates

The *empirical entropy* has been established by several papers [12, 23, 27] as a popular complexity measure for strings. While the classic notion of Shannon's entropy is a function of the source generating the input, the empirical entropy depends *only on the specific input string*. For this reason, it is naturally used to provide worst-case estimates on the output-size of compression algorithms.

In the rest of this paper, we will always denote with $T[1, n]$ the input string, whose symbols are drawn from the alphabet Σ of size σ. For each $c \in \Sigma$, we let n_c be the number of occurrences of c in T. The zero-th order *empirical entropy* of T is defined as $H_0(T) = \frac{1}{|T|} \sum_{c \in \Sigma} n_c \log \frac{n}{n_c}$, where it is assumed that all logarithms are taken to the base 2 and $0 \log 0 = 0$. It is well known that H_0 is the maximum compression one can achieve using a uniquely decodable code in which a fixed codeword is assigned to each alphabet symbol. In particular, the so-called *zero-th order statistical compressors* (such as Huffman or Arithmetic [33]) achieve an output size which is very close to this bound. However, they require to know information about frequencies of input symbols (called the *model* of the source). Those frequencies can be either known in advance (*static* model) or computed by examining the input text (*semistatic*

model).[3] In both cases the model must be stored in the compressed file to be used by the decompressor. In the following we will bound the compressed size achieved by zero-th order compressors over T by $|C_0(T)| \leq \lambda |T| H_0(T) + f_0(|T|, \sigma)$ bits, where λ is a positive constant and $f_0(|T|, \sigma)$ is a function including the extra costs of encoding the source model and/or other inefficiencies of C. We will assume that the function $f_0(|T|, \sigma)$ can be computed in constant time given $n = |T|$ and σ. As an example, for Huffman $f_0(n, \sigma) = O(\sigma \log \sigma) + n$ bits and $\lambda = 1$, and for Arithmetic $f_0(n, \sigma) = O(\sigma \log n)$ bits and $\lambda = 1$.

The distinctive feature of zero-th order compressors is that the input symbols are *independently encoded*, thus no advantage is taken of the potential statistical dependencies between consecutive symbols. Such dependencies are exploited by *higher order compressors* to achieve considerably better compression. In order to analyze the performances of these compressors it is necessary to refine the notion of entropy by taking into account of the correlations among a symbol and its *context*, that is the sequence of symbols immediately preceding it. For any substring u of length k, we denote by u_T the string of single symbols following the occurrences of u in T, taken from left to right. For example, if $T = \texttt{mississippi}$ and $u = \texttt{si}$, we have $u_T = \texttt{sp}$ since the two occurrences of $\texttt{si}$ in T are followed by the symbols $\texttt{s}$ and $\texttt{p}$, respectively. The k-th order *empirical* entropy of T is defined as $H_k(T) = \frac{1}{|T|} \sum_{u \in \Sigma^k} |u_T| H_0(u_T)$. This quantity provides a *lower bound* to any compressor that encodes each symbol with a codeword depending on the symbol itself and on the k immediately preceding symbols.

Example 1 Let $T = \texttt{mississippi}$. For $k = 1$ it is $\texttt{i}_T = \texttt{mssp}$, $\texttt{s}_T = \texttt{isis}$, $\texttt{p}_T = \texttt{ip}$. Hence,

$$H_1(T) = \frac{4}{11} H_0(\texttt{mssp}) + \frac{4}{11} H_0(\texttt{isis}) + \frac{2}{11} H_0(\texttt{ip}) = \frac{6}{11} + \frac{4}{11} + \frac{2}{11} = \frac{12}{11}.$$

Not surprisingly, we have that $H_k(T) \geq H_{k+1}(T)$ for any $k \geq 0$. Recently (see e.g. [12, 13, 16, 20, 23, 26, 27] and references therein) several authors have provided *upper bounds* in terms of $H_k(T)$ for sophisticated compressors, such as `gzip` [23], `bzip2` [12, 22, 27], and `ppm`. These bounds have the form $|C(T)| \leq \lambda |T| H_k(T) + f_k(|T|, \sigma)$, which is indeed a generalization of the one above used for zero-order compressors (with $k = 0$) where λ is a positive constant and $f_k(|T|, \sigma)$ is a function including the extra-cost of encoding the source model and/or other inefficiencies of C. We will say that C is a *k-th order compressor*, for a particular choice of k, whenever its output-length can be closely approximated by an entropy-based bound having the form above. The smaller are λ and $f_k()$, the better is the compressor C. As an example, the bound of the compressor in [26] has $\lambda = 1$ and $f(|T|, \sigma) = O(\sigma^{k+1} \log |T| + |T| \frac{\log \sigma \log \log |T|}{\log |T|})$.

In our approach, we will use entropy-based upper bounds for the estimation of $|C(T[i, j])|$. Clearly, this will not be enough to achieve a fast DP-based algorithm

[3] The frequencies can be also learned and deployed in the compression process as the input text is scanned (*dynamic* model [33]). This case will be dealt with in Sect. 9.

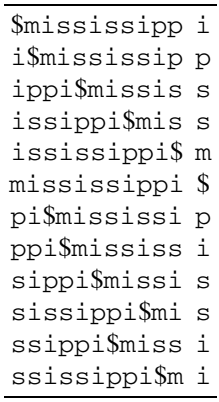

Fig. 1 Burrows-Wheeler Transform of the word `mississippi`. The last column of the conceptual matrix is the transformed string `ipsm$pissii`

for our optimal-partitioning problem. We cannot re-compute from scratch those estimates for every substring $T[i, j]$ of T, being them $\Theta(n^2)$ in number. So we will show some structural properties of our problem (Sect. 4) and introduce novel technicalities (Sects. 5–6) that will allow us to compute $H_k(T[i, j])$ only on a *reduced* subset of T's substrings, taking $O(n \log_{1+\varepsilon} n)$ time and $O(n)$ space overall.

2.2 The Burrows-Wheeler Transform

The Burrows-Wheeler transform of an input text T ($\texttt{BWT}(T)$ for short [6]) is computed via three steps: (1) append to the end of T a special symbol \$ smaller than any other symbol in Σ; (2) form a *conceptual* matrix M whose rows are the cyclic shifts of the string $T\$$, sorted in lexicographic order; (3) construct the transformed text $\texttt{BWT}(T)$ by taking the last column of M. See Fig. 1 for an example.

The Burrows-Wheeler transform is a permutation of $T\$$, and in [6] Burrows and Wheeler proved that it is possible to recover T from $\texttt{BWT}(T)$ in $O(|T|)$ time. The property that makes the `BWT` a powerful text compression tool is the following: for each substring u of T, the symbols preceding u in T form a substring of $\texttt{BWT}(T)$. This is a consequence of the fact that all the rows of the conceptual matrix prefixed by u appear consecutively in the lexicographic order. Thus, when applied to input exhibiting higher order correlations (such as natural language texts), the `BWT` is likely to generate a *locally homogeneous* string, consisting of the concatenation of several substrings made up of few distinct symbols. To take advantage of this property, compressors based on the `BWT` process the string $\texttt{BWT}(T)$ using a technique called *Move-To-Front encoding* [2] (`MTF`). `MTF` encodes each symbol with the number of distinct symbols encountered since its previous occurrence. The produced string has the same length as $\texttt{BWT}(T)$ and, if $\texttt{BWT}(T)$ is locally homogeneous, the `MTF`-string mainly consists of small integers which can be highly compressed by means of simple statistical encoders like Huffman or Arithmetic coding, possibly preceded by the run-length encoding (`RLE`) of runs of equal integers. The combination of the steps `BWT`, `MTF` and `RLE` followed by a zero-th order statistical compressor is the well-known Block-Sorting compression algorithm [9], which is at the basis of powerful text compressors such as `bzip2`.

Our approach to the optimal partitioning problem can be clearly applied to `bzip2`-like compressors. However, since their entropy-based estimates may result far from the real output-size (see e.g. [15]), we introduce in Sect. 7 a novel approach that computes *efficiently the exact output-size* of most `BWT`-based compressors, thus avoiding the use of (entropy-based) estimators for it.

2.3 Textual Compression Boosting

Compression Boosting is a technique that improves the compression performance of a wide class of (poor) data compressors. The works of [12, 18] tackled the question of compression boosting by using an instantiation of the PPC-paradigm in which the permuting step is implemented via the `BWT`. The net result was a tool for boosting the performance of a simple zero-th order base compressor C from bounds in H_0 to bounds in H_k, simultaneously over all $k \geq 0$. In particular, the booster of [12] returns a compressor which has essentially the same time/space complexity of the boosted (base-)compressor. All the boosting techniques proposed in the literature start from the next lemma, which links together the Burrows-Wheeler Transform, the empirical entropy and the optimal partitioning problem:

Lemma 1 [18] *For any string S and any positive integer k, there exists a partition $\hat{S}_1\hat{S}_2\ldots\hat{S}_f$ of the string $\texttt{BWT}(S)$ such that $|S|H_k(S) = \sum_{i=1}^{f} |\hat{S}_i|H_0(\hat{S}_i)$.*

This lemma suggests a new compression algorithm, denoted hereafter by $\texttt{BWT}^C_{OPT}$, which compresses the input text T via the following three basic steps:

(1) Compute $\texttt{BWT}(T)$;
(2) Optimally partition $\texttt{BWT}(T)$ with respect to the base compressor C;
(3) Separately compress with C each piece of the partition, and finally concatenate the results in output.

The next result is a consequence of Lemma 1:

Theorem 1 [18] *Let C be a compressor such that $|C(x)| \leq \lambda|x|H_0(x) + \mu|x| + c$, where λ, μ and c are non-negative values. Then, for all $k \geq 0$, the output-size in bits of the compressor $\texttt{BWT}^C_{OPT}$ applied to the string T is bounded by $\lambda|T|H_k(T) + \mu|T| + O(c\sigma^k)$.*

For instance, if C is the classic Huffman encoder, we have $\lambda = \mu = 1$ and $c = O(\sigma \log \sigma)$ and therefore $|\texttt{BWT}^C_{OPT}(T)| \leq |T|H_k(T) + |T| + O(\sigma^{k+1} \log \sigma)$. Theorem 1 thus reduces the problem of boosting a base compressor C to that of computing an optimal partitioning of the BW-transformed text, as required in step (2). Up to now, no efficient procedure is known for implementing this task. Ferragina et al. [12] (and subsequently many other authors, see e.g. [13, 16, 26]) proposed a partitioning technique which is not optimal but, nonetheless, achieves the k-th order entropy bound stated in Theorem 1. The main idea of [12] is to focus on a restricted family of partitions of the `BWT`. Let **ST** denote the suffix tree of the input text $T\$$. Any node u of **ST** has *implicitly associated* a substring of $T\$$, given by the concatenation of the

edge labels on the downward path from the root of **ST** to u. In that implicit association, the leaves of **ST** correspond to the suffixes of $T\$$. Assume that the suffix tree edges are sorted lexicographically. Since each row of the BWT matrix is prefixed by one suffix of $T\$$ and rows are lexicographically sorted, the i-th leaf (counting from the left) of the suffix tree corresponds to the i-th row of the BWT matrix. Associate to the i-th leaf of **ST** the i-th symbol of $\mathtt{BWT}(T)$.

For any suffix tree node u, let $\hat{T}\langle u\rangle$ denote the substring of $\mathtt{BWT}(T)$ obtained by concatenating, from left to right, the symbols associated to the leaves descending from node u. Of course $\hat{T}\langle root(\mathbf{ST})\rangle = \mathtt{BWT}(T)$. A subset L of **ST**'s nodes is called a *leaf cover* if every leaf of the suffix tree has a *unique* ancestor in L. Any leaf cover $L = \{u_1, \ldots, u_p\}$ naturally induces a partition of the leaves of **ST**, and because of the relationship between **ST** and the BWT matrix, this induces a partition of $\mathtt{BWT}(T)$ given by $\{\hat{T}\langle u_1\rangle, \ldots, \hat{T}\langle u_p\rangle\}$. For any leaf cover L, define its *cost* as the compression cost of the partition it induces on $\mathtt{BWT}(T)$, namely $\sum_{u\in L} |C(\hat{T}\langle u\rangle)|$. The optimal leaf cover is the one achieving the minimum cost.

The main contribution of [12] was to show that:

Theorem 2 *The optimal leaf cover can be computed in time $O(n\sigma)$ with a simple bottom-up visit of* **ST** *and, applying $\mathtt{BWT}^{C}_{OPT}$ on that partition, the final compressed output satisfies Theorem 1.*

The key contribution of Sect. 8 is to show that the partition of $\mathtt{BWT}(T)$ obtained via the optimal leaf-covering is not optimal but, nonetheless, achieves the nice k-th order-entropy bounds stated in Theorem 1. This is indeed a subtle point that we sustain by proposing an infinite class of strings for which the compression cost of any leaf-cover partition is far from the optimal one by a *multiplicative* factor $\Omega(\log n / \log\log n)$.

3 Approximating Dynamic Programming Solutions

In this section we will show how to efficiently approximate the solution of a Dynamic Programming Recurrence whenever its cost function satisfies a special (but indeed common!) property. We present this strategy in a general form since we believe that it could be used to speed up other algorithms based on the dynamic programming paradigm.

Let us consider a generic *one-dimensional Dynamic Programming Recurrence* of the form:

$$E[j] = \min_{1\le i<j} (E[i] + w(i, j)) \tag{1}$$

where $E[1]$ is equal to some constant c and $w()$ is a real-valued *cost function*[4] defined over the integer values $1 \le i < j \le n$. This type of dynamic programming recurrences has been extensively studied in the past since they have a large number of

[4] In this subsection we assume that $w()$ can be evaluated in constant time. This is a strong assumption since in many cases the efficient computation of its value is a difficult task that may require even sophisticated techniques. In fact, most of this paper is devoted to solve efficiently this task.

applications or can be used as a building block to solve more complex recurrences (see [17] and references therein). It is known that the näive algorithm that computes $E[\,]$ in quadratic time is optimal, if we do not make any assumption on the properties of the cost function $w()$. However, if $w()$ satisfies some special properties then speed ups are possible. For example, if the cost function satisfies the so-called *quadrangle inequality* (or its inverse) then the Recurrence 1 can be solved in linear time. The quadrangle inequality is defined as:

$$w(i_0, j_0) + w(i_1, j_1) \leq w(i_0, j_1) + w(i_1, j_0)$$

for any quadruple of indexes $1 \leq i_0 < i_1 < j_0 < j_1 \leq n$. Unfortunately, the quadrangle inequality is not much common and, in particular, it does not hold for the DP-recurrences arising in our optimal-partitioning problem.

So in this paper we make a step forward by showing that it is possible to compute an $(1+\varepsilon)$-approximation of the optimal solution to Recurrence 1 in $O(n \log_{1+\varepsilon} L)$ time and $O(n)$ space whenever the cost function $w()$ is *monotone*. Here ε is an arbitrary positive parameter fixed in advance, L is the largest cost assigned by $w()$ to any pair of indexes, and the *monotonicity property* states that for every pair of indexes $1 \leq i < j < n$, it is

$$w(i, j) \leq w(i, j+1) \quad \text{and} \quad w(i+1, j) \leq w(i, j).$$

It is easy to see that $w()$ can be monotone without satisfying the quadrangle inequality. In order to simplify the explanation of our solution it is convenient to resort to a well-known reduction [8] from the problem of solving Recurrence 1 to the problem of computing a single source shortest path (SSSP) over a particular directed acyclic graph (DAG) $\mathcal{G}$. This graph $\mathcal{G}$ has a vertex v_i for each entry of E and an edge connecting v_i to v_j whose cost is $w(i, j)$, for any pair of indexes i and j such that $i < j$. Clearly we have $E[n] = d_{\mathcal{G}}(v_1, v_n)$, where we denote with $d_{\mathcal{G}}(v_i, v_j)$ the shortest path distance between the vertices v_i and v_j. Notice that $\mathcal{G}$ has $\Theta(n^2)$ edges, so the reduction by itself does not improve the complexity of the näive algorithm.

In order to obtain a faster solution we design a *pruning* strategy that produces a subgraph $\mathcal{G}_\varepsilon$ such that:

1. the number of edges is significantly reduced, from $\Theta(n^2)$ to $O(n \log_{1+\varepsilon} L)$;
2. the shortest path distance in $\mathcal{G}_\varepsilon$ between its leftmost v_1 and rightmost v_n vertices increases by no more than a factor $(1+\varepsilon)$.

The pruned graph $\mathcal{G}_\varepsilon$ is constructed as the subgraph of $\mathcal{G}$ consisting of all edges (v_i, v_j) such that at least one of the following two conditions holds:

1. there exists a positive integer k such that $w(v_i, v_j) \leq (1+\varepsilon)^k < w(v_i, v_{j+1})$;
2. $j = n$.

In other words, since $w()$ is monotone, for each integer k we are keeping the edge of $\mathcal{G}$ that best approximates the value $(1+\varepsilon)^k$ from below. The edges of $\mathcal{G}_\varepsilon$ are called ε-*maximal* edges. We point out that each vertex of $\mathcal{G}_\varepsilon$ has at most $\log_{1+\varepsilon} L$ outgoing (ε-*maximal*) edges and, thus, the total size of $\mathcal{G}_\varepsilon$ is $O(n \log_{1+\varepsilon} L)$.

The following lemma states simple properties of shortest-path distances over $\mathcal{G}$ that will be useful in the proof of the main theorem of this subsection.

Lemma 2 *For any triple of indexes* $1 \le i \le j \le q \le n$ *we have*:

1. $d_{\mathcal{G}}(v_i, v_q) \ge d_{\mathcal{G}}(v_j, v_q)$;
2. $d_{\mathcal{G}}(v_i, v_j) \le d_{\mathcal{G}}(v_i, v_q)$.

Proof We prove only 1, since 2 is symmetric. It suffices by induction to prove the case $j = i + 1$. Let $(v_i, u_1)(u_1, u_2)\dots(u_{h-1}, u_h)$, with $u_h = v_q$, be a shortest path in $\mathcal{G}$ from v_i to v_q. Since $w()$ is monotone and by the fact that $i \le j$, it holds $w(v_i, u_1) \ge w(v_j, u_1)$. Therefore the cost of the path $(v_j, u_1)(u_1, u_2)\dots(u_{h-1}, u_h)$ is at most $d_{\mathcal{G}}(v_i, v_q)$, which proves the claim. □

The correctness of the pruning strategy relies on the following theorem:

Theorem 3 *If the cost function* $w()$ *is monotone, the shortest path in* $\mathcal{G}_\varepsilon$ *from* v_1 *to* v_n *has total cost upper bounded by* $(1+\varepsilon)\, d_{\mathcal{G}}(v_1, v_n)$.

Proof We prove a stronger assertion: $d_{\mathcal{G}_\varepsilon}(v_i, v_n) \le (1+\varepsilon)\, d_{\mathcal{G}}(v_i, v_n)$ for any index $1 \le i \le n$. This is clearly true for i, because in that case the distance is 0. Now let us inductively consider the shortest path π in $\mathcal{G}$ from v_k to v_n and let $(v_k, v_{t_1})(v_{t_1}, v_{t_2})\dots(v_{t_h} v_n)$ be its edges. By the definition of ε-maximal edge, it is possible to find an ε-maximal edge (v_k, v_r) with $r \ge t_1$, such that $w(v_k, v_r) \le (1+\varepsilon)\, w(v_k, v_{t_1})$. By Lemma 2, $d_{\mathcal{G}}(v_r, v_n) \le d_{\mathcal{G}}(v_{t_1}, v_n)$; and by induction we know that $d_{\mathcal{G}_\varepsilon}(v_r, v_n) \le (1+\varepsilon)\, d_{\mathcal{G}}(v_r, v_n)$.

We are interested in upper bounding the cost of the shortest path in $\mathcal{G}_\varepsilon$ from v_k to v_n. By definition, this cost is smaller than the cost of any other path in $\mathcal{G}_\varepsilon$ that connects these two vertices: so let us take the path starting with the edge (v_k, v_r) (which is ε-maximal) and then proceed with the shortest path in $\mathcal{G}_\varepsilon$ from v_r to v_n. This path has cost $w(v_k, v_r) + d_{\mathcal{G}_\varepsilon}(v_r, v_n)$, which can be upper bounded by using the previous inequalities as $(1+\varepsilon)\, w(v_k, v_{t_1}) + (1+\varepsilon) d_{\mathcal{G}}(v_r, v_n) \le (1+\varepsilon)(w(v_k, v_{t_1}) + d_{\mathcal{G}}(v_{t_1}, v_n))$. The last value is exactly $(1+\varepsilon)\, d_{\mathcal{G}}(v_k, v_n)$ by definition of π. □

Combining the reduction technique of this section with the statement of Theorem 3 we immediately obtain the following corollary:

Corollary 1 *If the cost function* $w()$ *is monotone, for any positive value* ε, *the shortest-path distance in* $\mathcal{G}_\varepsilon$ *from* v_1 *to* v_n *is an* $(1+\varepsilon)$*-approximation* $E_\varepsilon[n]$ *of* $E[n]$, *where* $E[n]$ *is the solution of Recurrence* 1, *namely,* $E[n] \le E_\varepsilon[n] \le (1+\varepsilon) E[n]$.

Corollary 1 implies that an approximate solution to Recurrence 1 can be obtained by performing an SSSP-computation over a graph whose size is significantly smaller than $\mathcal{G}$: namely, $O(n \log_{1+\varepsilon} L)$ instead of $\Theta(n^2)$, where L is the maximum edge weight. However, even though the SSSP-computation over a DAG can be performed in time proportional to its number of edges, we still have the problem of efficiently generating the graph $\mathcal{G}_\varepsilon$. In fact, the obvious solution which takes $\mathcal{G}$ and discards the edges not belonging to $\mathcal{G}_\varepsilon$ takes $\Theta(n^2)$ time and space. More subtly, even if we are able to produce $\mathcal{G}_\varepsilon$ without passing through $\mathcal{G}$, we would need $\Omega(n \log_{1+\varepsilon} L)$ space to store $\mathcal{G}_\varepsilon$, and this would make the space occupancy of the solution super-linear in

the input size n. Therefore the generation of $\mathcal{G}_\varepsilon$ in efficient time and space is a *non trivial* task.

The remaining part of the paper is devoted to devise efficient solutions for this problem. Our main achievement will be an algorithm to generate $\mathcal{G}_\varepsilon$ in optimal $O(n)$ space and in time that depends linearly on the number of ε-maximal edges, in the case where $w()$ is the cost function deployed in the optimal partitioning problem. Thus, we will finally derive the following result:

Theorem 4 *If the cost function $w()$ is monotone, we can compute in $O(n \log_{1+\varepsilon} L)$ time and $O(n)$ space an $(1+\varepsilon)$-approximation of $E[n]$, where $E[n]$ is the solution of Recurrence 1, ε is an arbitrary positive real value fixed in advance and L is the largest cost assigned by the cost function $w()$.*

4 Optimal Partitioning Problem

It is easy to notice that the optimal partitioning problem stated in Sect. 1 can be solved with Recurrence 1 by setting $w(i, j) = |C(T[i, j-1])|$ where $|C(T[i, j-1])|$ denotes the size in bits of the substring $T[i, j-1]$ compressed by C. The corresponding DAG, denoted $\mathcal{G}(T)$, has a vertex v_i for each text position i of T, plus an additional vertex v_{n+1} marking the end of the text, and an edge connecting vertex v_i to vertex v_j with associated the cost $w(v_i, v_j) = |C(T[i, j-1])|$. Notice that the edge lands to the text symbol that follows the substring "compressed" by that edge. In what follows we assume that $w()$ is monotone, so that we can resort to our pruning strategy. This assumption holds for almost any realistic compressor C, because it simply assumes that compressing any string s by C produces an output which is longer than compressing any prefix or suffix of s. As a result, we can apply Theorem 3 and thus obtain a $(1+\varepsilon)$-approximation of the optimal partition of T from the computation of the SSSP in $\mathcal{G}_\varepsilon(T)$ from v_1 to v_{n+1}. This can be easily done in $O(|\mathcal{G}_\varepsilon(T)|) = O(n \log_\varepsilon n)$ time since $\mathcal{G}_\varepsilon(T)$ is a DAG [8], by making a single pass over its vertices according to their enumeration.

However, the time/space efficient construction of $\mathcal{G}_\varepsilon(T)$ is non trivial for three main reasons. First, the original graph $\mathcal{G}(T)$ contains $\Omega(n^2)$ edges, so we cannot construct $\mathcal{G}_\varepsilon(T)$ by pruning $\mathcal{G}(T)$'s edges via the explicit check of whether they are ε-maximal or not. Second, we cannot compute the cost of an edge (v_i, v_j) by executing $C(T[i, j-1])$ *from scratch*, since this would require time linear in the substring length, and thus $\Omega(n^3)$ time over all T's substrings. Third, we cannot materialize $\mathcal{G}_\varepsilon(T)$ (i.e. its adjacency lists) because it consists of $\Theta(n \text{ polylog}(n))$ edges, and thus its space occupancy would be super-linear in the input size $n = |T|$.

The rest of this section is therefore devoted to design an algorithm which overcomes the three limitations above. The specialty of our algorithm consists of materializing $\mathcal{G}_\varepsilon(T)$ on-the-fly, as its vertices are examined during the SSSP-computation, spending no more than poly-logarithmic time per edge. The actual time complexity per edge will depend on the entropy-based cost function we will use to estimate $|C(T[i, j-1])|$ (see Sect. 2) and on the dynamic data structure we will deploy to compute this estimation.

The key tool we use to make a fast estimation of the edge costs is a dynamic data structure built over the input text T and requiring $O(|T|)$ space. We state the main properties of this data structure in an abstract form, in order to set-up a general framework for solving our problem; in the next sections we will then provide several implementations of this data structure depending on the context of use (e.g. zero-th order compressor, k-th order compressor, BWT-based compressor, etc.) and thus obtain real time/space bounds for our solutions.

So let us assume to have a dynamic data structure that maintains a set of *sliding windows* over T denoted by $w_1, w_2, \ldots, w_{\log_{1+\varepsilon} n}$. The sliding windows are substrings of T which start at the same text position l but have different lengths: namely, $w_i = T[l, r_i]$ and $r_1 \leq r_2 \leq \ldots \leq r_{\log_{1+\varepsilon} n}$. The data structure must support the following three operations:

1. Remove() moves the start-position l of all windows one position to the right (i.e. $l + 1$);
2. Append(w_i) moves the end-position of the window w_i one position to the right (i.e. $r_i + 1$);
3. Size(w_i) computes and returns the value $|C(T[l, r_i])|$.

The crucial point is that this data structure is enough to generate the ε-maximal edges of $\mathcal{G}(T)$ via a single pass of T using $O(|T|)$ optimal space. More precisely, let v_l be the vertex of $\mathcal{G}(T)$ currently examined by our SSSP computation, and thus l is the current position reached by our scan of T. We maintain the following invariant: the sliding windows correspond to all ε-maximal edges going out from v_l, that is, the edge (v_l, v_{1+r_t}) is the ε-maximal edge[5] satisfying $w(v_l, v_{1+r_t}) \leq (1+\varepsilon)^t < w(v_l, v_{1+(r_t+1)})$. Initially all indexes are set to 0. To maintain the invariant, when the text scan advances to the next position $l + 1$, we call operation Remove() to increment index l and, for each $t = 1, \ldots, \log_{1+\varepsilon}(n)$, we repeatedly call operation Append(w_t) until we find the largest r_t such that Size(w_t) $= w(v_l, v_{1+r_t}) \leq (1+\varepsilon)^t$. The key issue here is that Append and Size are paired so that our data structure should take advantage of the rightward sliding of r_t for computing $w(v_l, v_{1+r_t})$ efficiently. Just one symbol is entering w_t to its right, so we need to deploy this fact for making the computation of Size(w_t) fast (given its previous value). Here comes into play the second contribution of our paper that consists of adopting the entropy-bounded estimates for the compressibility of a string, mentioned in Sect. 2, to estimate indeed the edge costs Size(w_t) $= |C(w_t)|$. This idea is crucial because we will be able to show that these functions do satisfy some structural properties that admit a *fast incremental computation*, as the one required by the execution of the instruction-pair Append + Size.

These issues will be detailed in the following sections, here we just state that, overall, the SSSP computation over $\mathcal{G}_\varepsilon(T)$ takes $O(n)$ calls to operation Remove, and $O(n \log_{1+\varepsilon} n)$ calls to operations Append and Size.

Theorem 5 *If we have a dynamic data structure occupying $O(n)$ space and supporting operation* Remove *in time $L(n)$, and operations* Append *and* Size *in*

[5] Recall that an edge identifies a substring which ends one character before its landing position, hence the $+1$ with r_t.

time $R(n)$, then we can compute the shortest path in $\mathcal{G}_\varepsilon(T)$ from v_1 to v_{n+1} taking $O(n\,L(n) + (n \log_{1+\varepsilon} n)\,R(n))$ time and $O(n)$ space.

5 On Zero-th Order Compressors

In this section we explain how to implement the data structure of Theorem 5 whenever the base compressor C is a zero-th order compressor, and thus H_0 is used to provide a bound to the (compression) cost of the edges of $\mathcal{G}(T)$, as discussed in Sect. 2. The key point is to show how to efficiently compute $\texttt{Size}(w_i)$ as the sum of $|T[l, r_i]|H_0(T[l, r_i]) = \sum_{c\in\Sigma} n_c \log((r_i - l + 1)/n_c)$ plus $f_0(r_i - l + 1, |\Sigma_{T[l,r_i]}|)$, where n_c is the number of occurrences of symbol c in $T[l, r_i]$ and $|\Sigma_{T[l,r_i]}|$ denotes the number of different symbols in the substring $T[l, r_i]$.

The first solution we are going to present is very simple and uses $O(\sigma)$ space per window, hence $O(\sigma \log_{1+\varepsilon} n)$ space overall. The idea is the following: for each window w_i we keep in memory an array of counters $A_i[c]$ indexed by symbol c in Σ. At any step of our algorithm, the counter $A_i[c]$ stores the number of occurrences of symbol c in $T[l, r_i]$. For any window w_i, we also use a variable E_i that stores the value $\sum_{c\in\Sigma} A_i[c] \log A_i[c]$. We have:

$$|T[l, r_i]|\, H_0(T[l, r_i]) = (r_i - l + 1) \log(r_i - l + 1) - E_i. \tag{2}$$

Therefore, if we know the value of E_i, we can answer a query $\texttt{Size}(w_i)$ in constant time. So we are left with showing how to modify the E's value as a result of a `Remove` or `Append` operation. This can be done as follows:

1. `Remove`: For each window w_i, we subtract from the appropriate counter and from the variable E_i the contribution of the symbol $T[l]$ which has been evicted from the window. That is, we decrease $A_i[T[l]]$ by one, and update E_i by subtracting $(A_i[T[l]] + 1) \log(A_i[T[l]] + 1)$ and then summing $A_i[T[l]] \log A_i[T[l]]$. Finally we set $l = l + 1$.
2. $\texttt{Append}(w_i)$: We add to the appropriate counter and variable E_i the contribution of the symbol $T[r_i + 1]$ which has been appended to window w_i. That is, we increase $A_i[T[r_i + 1]]$ by one, then we update E_i by subtracting $(A[T[r_i + 1]] - 1) \log(A[T[r_i + 1]] - 1)$ and summing $A[T[r_i + 1]] \log A[T[r_i + 1]]$. Finally we set $r_i = r_i + 1$.

This way, operation `Remove` requires constant time per window, hence $O(\log_{1+\varepsilon} n)$ time overall. $\texttt{Append}(w_i)$ takes constant time. The space required by all counters A_i is $O(\sigma \log_{1+\varepsilon} n)$. Unfortunately, this can be too much if this solution is used as the basic-block for computing the k-th order entropy of T as we will do in Sect. 6. In that case we would get $\min(\sigma^{k+1}, n) \log_{1+\varepsilon} n$ space, which may be superlinear in n depending on σ and k.

Therefore, the rest of this section is devoted to provide an implementation of our dynamic data structure that takes the same query time above for the three operations—`Remove`, `Append` and `Size`—but within $O(n)$ space, thus resulting independent from σ and k. The new solution still uses E_i's value for computing $\texttt{Size}(w_i)$ (according to Eq. 2), but the counters A_i previously used to determine

E_i are computed *on-the-fly* by exploiting the fact that all windows share the same starting position l. In particular we introduce two arrays:

- An array $B[1, \sigma]$ indexed by symbols, whose entry $B[c]$ stores the number of occurrences of symbol c in $T[1, l-1]$.
- An array $R[1, n]$ indexed by text positions, whose entry $R[j]$ stores the number of occurrences of symbol $T[j]$ in $T[1, j]$.

The number of elements in both B and R is $n + \sigma = O(n)$. These two arrays are enough to evaluate the value E_i which, in turn, is enough to estimate H_0 (see Eq. 2), as operations `Remove` and `Append` are executed. Let us see how.

First of all we notice that R can be computed via a scan of T in $O(n)$ time, and its values are independent of `Remove-Append` operations. On the other hand $B[1, \sigma]$ depends on the starting position l of the windows w_i but not on its ending positions, so its content is not influenced by `Append` and can be trivially changed after the `Remove` operation by increasing $B[T[l]]$ of one unit. So, hereafter, we assume to have up-to-date all values of B and R as operations `Remove` and `Append` are executed.

Let us now show how to correctly update the value E_i after each operation `Append` and `Remove`, given that B and R are available. Following the scheme indicated in items 1 and 2 before, we notice that `Append`(w_i) needs to compute $A_i[T[r_i + 1]]$, which is the number of occurrences of $T[r_i + 1]$ in $T[l, r_i + 1]$. This is given by $R[r_i + 1] - B[T[r_i + 1]]$, thus it is computable in $O(1)$ time.

Conversely, the execution of `Remove` induces an update of E_i which is more involved because it requires to evaluate the value of $A_i[T[l]]$ for each window w_i. Similarly as before, we could compute each of these values as $R[t] - B[T[l]]$, where t is the last occurrence of symbol $T[l]$ in $T[l, r_i]$. The problem with this formula is that we do not know the position t. We solve this issue by resorting to a doubly linked list L_c for each symbol c. The list L_c links together the last occurrences of c in all those windows, ordered by increasing position. Notice that a position j may be the last occurrence of symbol $T[j]$ for different, but consecutive, windows given that the windows w_i have increasing length. In this case we force this *shared* position to occur in $L_{T[j]}$ just once. By construction, these lists are sufficient to derive the value of $A_i[T[l]]$: for each position $t \in L_{T[l]}$ compute $R[t] - B[T[l]]$. Once we have $A_i[T[l]]$, for all windows w_is, we update all E_i's as explained in the above item 1. This process takes $O(\log_{1+\varepsilon} n)$ time, because $|L_{T[l]}| \leq \log_{1+\varepsilon} n$, and uses $O(n)$ space, because the number of elements in all the lists L_c is bounded by the text length.

It remains to explain how to keep lists L correctly updated after a `Remove()` or an `Append`(w_i) operation. First, notice that only one list may change because just one symbol enters/exits the windows w_i. More precisely, `Remove` has possibly to remove position l from list $L_{T[l]}$. This change is easy because, if that position is in the list, given that $T[l]$ is the last occurrence of that symbol in w_1 (recall that all the windows start at position l, and are kept ordered by increasing ending position), thus l must be the head of $L_{T[l]}$. So we find it efficiently. The change on the Ls induced by `Append`(w_i) is more involved. Since the ending position of w_i is moved to the right, position $r_i + 1$ becomes the last occurrence of symbol $T[r_i + 1]$ in w_i.

Thus, this position must be inserted in $L_{T[r_i+1]}$ in its correct (sorted) order, if it is not present yet. Obviously, we can do that in $O(\log_{1+\varepsilon} n)$ time by scanning the whole list, but this is too much. So we show how to spend only constant time. Let p be the rightmost occurrence of the symbol $T[r_i+1]$ in $T[0, r_i]$.[6] If $p < l$, then $r_i + 1$ must be inserted in the front of $L_{T[r_i+1]}$ and we have done. In fact, $p < l$ implies that there is no occurrence of $T[r_i + 1]$ in $T[l, r_i]$ and, thus, no position can precede $r_i + 1$ in $L_{T[r_i+1]}$. Otherwise (i.e. $p \geq l$), we have that p is in $L_{T[r_i+1]}$, because it is the last occurrence of symbol $T[r_i + 1]$ for some window w_j with $j \leq i$. We observe that if $w_j = w_i$, then p must be replaced by $r_i + 1$ which is now the last occurrence of $T[r_i + 1]$ in w_i; otherwise $r_i + 1$ must be inserted after p in $L_{T[r_i+1]}$ because p is still the last occurrence of this symbol in the window w_j. We can decide which one is the correct case by comparing p and r_{i-1} (i.e., the ending position of the preceding window w_{i-1}). In any case, the list is kept updated in constant time.

We have therefore proved the following:

Lemma 3 *Let $T[1, n]$ be a text drawn from an alphabet of size $\sigma = \text{poly}(n)$. If we estimate* `Size()` *via zero-th order entropy (as detailed in Sect. 2), then we can design a dynamic data structure that takes $O(n)$ space and supports the operations* `Remove` *in $R(n) = O(\log_{1+\varepsilon} n)$ time, and* `Append` *and* `Size` *in $L(n) = O(1)$ time.*

Combining Theorem 5 and Lemma 3 we obtain:

Theorem 6 *Given a text $T[1, n]$ drawn from an alphabet of size $\sigma = \text{poly}(n)$, we can find an $(1 + \varepsilon)$-optimal partition of T with respect to a zero-th order compressor in $O(n \log_{1+\varepsilon} n)$ time and $O(n)$ space, where ε is any positive constant.*

The case of zero-th order compressors is interesting in practice because *Huffword* is one of the standard choices for the storage of Web pages [33]. In this case Σ consists of the distinct words appearing in the Web-page collection, so σ is large, and thus our technical improvement that made our solution independent on σ is particularly relevant in this case. Section 8 will deploy this result also to obtain a fast approximation of the optimal partition of $\texttt{BWT}(T)$, and thus *optimize* the compression booster of [12].

6 On k-th Order Compressors

In this section we make one step further and consider the more powerful k-th order compressors, for which do exist H_k bounds for estimating the size of their compressed output (see Sect. 2). Here $\texttt{Size}(w_i)$ must compute $|C(T[l, r_i])|$ which is estimated by the k-th order compressibility of $T[l, r_i]$, namely $(r_i - l + 1)H_k(T[l, r_i]) + f_k(r_i - l + 1, |\Sigma_{T[l,r_i]}|)$, where $\Sigma_{T[l,r_i]}$ denotes the number of different symbols in $T[l, r_i]$.

[6] Notice that we can precompute and store the last occurrence of symbol $T[j + 1]$ in $T[1, j]$ for all js in linear time and space.

Let us denote with $T_q[1, n-q]$ the text whose i-th symbol $T[i]$ is equal to the q-gram $T[i, i+q-1]$. Actually, we can remap the symbols of T_q to integers in $[1, n]$, because the number of distinct q-grams occurring in T_q is less than n. Thus T_q's symbols take $O(\log n)$ bits and T_q can be stored in $O(n)$ space. This remapping takes linear time and space, whenever σ is polynomial in n, and it does not change the zero-th order entropy of T_q.

It is well known that the k-th order entropy of a string can be expressed as the difference between the zero-th order entropy of its $k+1$-grams and its k-grams (see definition Sect. 2). This suggests to use the solution of the previous section for computing the zero-th order entropy of the appropriate substrings of T_{k+1} and T_k. More precisely, we use two instances of the data structure of Theorem 6 (one for T_{k+1} and one for T_k), which are kept *synchronized* in the sense that, when operations are performed on one data structure, then they are also executed on the other.

Lemma 4 *Let $T[1, n]$ be a text drawn from an alphabet of size $\sigma = \text{poly}(n)$. If we estimate* `Size()` *via k-th order entropy (as detailed in Sect. 2), then we can design a dynamic data structure that takes $O(n)$ space and supports the operations* `Remove` *in $R(n) = O(\log_{1+\varepsilon} n)$ time, and* `Append` *and* `Size` *in $L(n) = O(1)$ time.*

Combining Theorem 5 and Lemma 4 we obtain:

Theorem 7 *Given a text $T[1, n]$ drawn from an alphabet of size $\sigma = \text{poly}(n)$, we can find an $(1+\varepsilon)$-optimal partition of T with respect to a k-th order compressor in $O(n \log_{1+\varepsilon} n)$ time and $O(n)$ space, where ε is any positive constant.*

We point out that this result applies also to the practical case in which the base compressor C can process at once a maximum block-size B (this is the typical scenario for `gzip`, `bzip2`, etc.). In this situation, we can restrict the set of ε-maximal edges to the ones that cover no more than B vertices. Given that the maximal cost of a B-long substring of T is $O(B \log \sigma)$, i.e. the case of un-compressible substring, the total number of ε-maximal edges outgoing from a vertex is $O(\log_{1+\varepsilon}(B \log \sigma))$ (see definition in Sect. 3). Consequently, the time performance of our solution reduces to $O(n \log_{1+\varepsilon}(B \log \sigma))$ in this case.

7 On `BWT`-Based Compressors

As we mentioned in Sect. 2 we know entropy-bounded estimates for the output size of `BWT`-based compressors. So we could apply Theorem 7 to determine the optimal partitioning of T for such types of compressors. However, equally known [15] is that such compression-estimates are rough in practice and thus of poor use.

In this section, we propose a solution to the optimal partitioning problem for `BWT`-based compressors that introduces a $\Theta(\sigma \log n)$ slowdown in the time complexity of Theorem 7, but with the advantage of computing the $(1+\varepsilon)$-optimal solution wrt the *real compressed size*, thus without any estimation by entropy-cost functions. When σ is small, this slowdown results negligible.

In order to achieve this result, we need to address a slightly different (but yet interesting) problem defined as follows. Assume that the input string T has the form $S[1]\#_1 S[2]\#_2 \ldots S[m]\#_n$ where each $S[i]$ is a text (called *page*) drawn from an alphabet Σ, and $\#_1, \#_2, \ldots, \#_n$ are special symbols greater than any other symbol of Σ. A partition of T must be page-aligned, that is it must form *groups of contiguous pages* $S[i]\#_i \ldots S[j]\#_j$, denoted also $S[i, j]$. Our aim is to find a page-aligned partition whose compression cost (as defined in Sect. 1) is at most $(1+\varepsilon)$-times the minimum possible compression cost, for any fixed $\varepsilon > 0$. We notice that this problem generalizes the table partitioning problem [5], since we can assume that $S[i]$ is a column of the table but of variable length.

For simplicity of exposition, we assume a "simplified" Block-Sorting algorithm that does not use the `RLE` encoding step.[7] Our solution deploys a close analog of Theorem 5 with the only difference that the windows $w_1, w_2, \ldots, w_m$ subject to the operations `Size`, `Append` and `Remove` are groups of *contiguous pages* of the form $w_i = S[l, r_i]$.

It goes without saying that the data structure of Theorem 5 could be implemented by using any known solution that dynamically maintains a string compressed with a `BWT`-based compressor under insertions and deletions of symbols and apply it onto the windows w_i (see e.g. [11] and references therein). Unfortunately, these solutions do not fit our context for two reasons: (1) their underlying compressor is significantly different from the scheme above; (2) in the worst case, they would spend linear space per window yielding a super-linear space complexity overall. Conversely, we propose in this section to evaluate the *exact* compressed output-size of the `BWT`-based algorithm applied on each window w_i by keeping the frequency distribution of the integers in the string $w_i' = \texttt{MTF}(\texttt{BWT}(w_i))$, simultaneously over all windows w_i.[8] This information is enough to compute in $O(\sigma)$ time the final compressed output-size of the `BWT`-based compressors because they typically use as final compression step an Huffman or Arithmetic statistical encoder over the `MTF`-output [33].

Recall that $\texttt{BWT}(T)$ is a permutation of T; so we denote by $active_{[a,b]}$ all occurrences of symbols of $T[a, b]$ within $\texttt{BWT}(T)$. The next simple result will be useful:

Theorem 8 *There exists a data structure that takes $O(n)$ space and answers the following two queries in $O(\log n)$ time*:

- $\texttt{Prev}_c(I, h)$: *locate the last $active_{[a,b]}$ occurrence in $\texttt{BWT}(T)[0, h-1]$ of symbol c*;
- $\texttt{Next}_c(I, h)$: *locate the first $active_{[a,b]}$ occurrence in $\texttt{BWT}(T)[h+1, n]$ of symbol c*,

where $c \in \Sigma$, $I = [a, b]$ is a range of positions in T and h is a position in $\texttt{BWT}(T)$.

Proof This is achieved by a straightforward reduction to a classic geometric range-searching problem. Given a set of points $P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_p, y_p)\}$ from

[7] The reader is invited to exercise herself to manage `RLE` too.

[8] Recall that $\texttt{MTF}(s)$ turns the string s drawn from alphabet Σ, into a sequence of integers in the range $[1, \sigma]$.

the set $[n] \times [n]$ (notice that n can be larger than p), such that no pair of points share the same x- or y-coordinate, there exists a data structure [25] requiring $O(p)$ space and supporting the following two queries in $O(\log p)$ time:

- `rangemax`$([l, r], h)$: return among the points of P contained in $[l, r] \times [-\infty, h]$ the one with maximum y-value;
- `rangemin`$([l, r], h)$: return among the points of P contained in $[l, r] \times [h, +\infty]$ the one with minimum y-value.

Initially we compute the suffix array $sa_T[1, n]$ and the inverse suffix array $isa_T[1, n]$ of text T in $O(n \log \sigma)$ time. Then, for each symbol $c \in \Sigma$, we define P_c as the set of points $\{(i, isa_T[i + 1]) | \ T[i] = c\}$ and build the above geometric range-searching structure on P_c. It is easy to see that $\texttt{Prev}_c(I, h)$ can be computed in $O(\log n)$ time by calling `rangemax`(I, h) on the set P_c, and the same holds for $\texttt{Next}_c$ by using `rangemin` instead of `rangemax`. □

Now we are ready to introduce our solution to Theorem 5 which hinges on two key facts (whose proofs belongs to folklore!):

1. Since the pages are separated in T by distinct separators, inserting or removing one page into a window w does not alter the relative lexicographic order of the original suffixes of w. Therefore we can derive $\texttt{BWT}(w)$ from $\texttt{BWT}(T)$ by concatenating w's symbols in accordance with their order in the whole $\texttt{BWT}(T)$ (see [11] and references therein). This specifically means that we can work *implicitly* on $\texttt{BWT}(w)$ by making use of $\texttt{BWT}(T)$ and the span of text positions covered by w.
2. If string s' is obtained from string s by inserting or removing a symbol c into an arbitrary position, then $\texttt{MTF}(s')$ differs from $\texttt{MTF}(s)$ in at most σ integers. More precisely, if c' is the next occurrence in s of the newly inserted (or removed) symbol c, then the MTF has to be updated only in the integers that correspond to the first occurrence of some of the symbols of Σ that lie between c and c'.

As observed above, we need to estimate at each window w the frequency of the symbols in $w' = \texttt{MTF}(\texttt{BWT}(w))$. These symbols are integers in $[1, \sigma]$, so we maintain an array $F_w[1, \sigma]$ for the MTF-symbol frequencies in w. The update of F_w after a `Remove`() or an $\texttt{Append}(w)$ operation is difficult because pages are added/removed from the extremes of the window w and we cannot recompute $\texttt{MTF}(\texttt{BWT}(w))$ from scratch at each operation. In the following we will concentrate only on $\texttt{Append}(w)$ since `Remove` is symmetrical.

The idea to perform $\texttt{Append}(w)$, where $w = S[l, r]$, is to *conceptually insert* the symbols of the next page $S[r + 1]$ into $\texttt{BWT}(w)$ one at time from left to right. From item 1 above, the relative order of the symbols of $\texttt{BWT}(w)$ is preserved in $\texttt{BWT}(T)$, so it is more convenient to work *implicitly* with w's symbols in $\texttt{BWT}(T)$ by deploying the data structure of Theorem 8 built over the whole text T. Precisely, let c be one of the symbols of $S[r_i + 1]$ we have to *logically* insert in $\texttt{BWT}(w)$. We can compute the position (say, h) of this symbol in $\texttt{BWT}(T)$ by deploying the inverse suffix array of T, which can be constructed in advance taking $O(n \log \sigma)$ time and $O(n)$ space. Once we know position h, we have to determine what changes in $\texttt{MTF}(\texttt{BWT}(w))$ the insertion of c has produced and update F_w accordingly. From item 2 above, the insertion of

symbol c changes no more than σ symbols of $\texttt{MTF}(\texttt{BWT}(w))$. More precisely, let h_L be the last occurrence of c before position h in $\texttt{BWT}(w)$ and let h_F be the first occurrence of c after h in $\texttt{BWT}(w)$. Then the first occurrence of a symbol after h changes its $\texttt{MTF}$-encoding if and only if it occurs both in $\texttt{BWT}(w)[h_L, h]$ and in $\texttt{BWT}(w)[h, h_F]$. Otherwise, the new occurrence of c has no effect on the $\texttt{MTF}$-encoding of that symbol. Notice that h_L and h_F can be computed with queries $\texttt{Prev}_c$ and $\texttt{Next}_c$ in $O(\log n)$ time by Theorem 8. In order to correctly update F_w, we need to recover for each of the above symbols their old and new encodings. The first step consists of finding the last active occurrence before h of each symbols in Σ using $\texttt{Prev}$ queries. According to Theorem 8 this takes $O(\sigma \log n)$ time per symbol of the new page $S[r+1]$. Once we have these positions, we can recover the status of the $\texttt{MTF}$ list, denoted γ, before encoding c at position h. This is simply obtained by sorting the symbols of Σ ordered by decreasing position, in $O(\sigma \log \sigma)$ time. In the second step, for each distinct symbol that occurs in $\texttt{BWT}(w)[h_L, h]$, we find its first occurrence in $\texttt{BWT}(w)[h, h_F]$ (if any). Knowing γ and these occurrences sorted by increasing position, we can simulate the $\texttt{MTF}$ algorithm to find the old and new encodings of each those symbols, taking $O(\sigma)$ time.

Overall, $\texttt{Append}(w)$ takes $O(\sigma \log n)$ time per symbol of the page to be appended to w. Thus we have proven the following:

Theorem 9 *Given a sequence of texts of total length n and alphabet size $\sigma = \text{poly}(n)$, we can compute an $(1+\varepsilon)$-approximate solution to the optimal partitioning problem for a $\texttt{BWT}$-based compressor, in $O(n(\log_{1+\varepsilon} n)\, \sigma \log n)$ time and $O(n + \sigma \log_{1+\varepsilon} n)$ space.*

8 A Nearly-Optimal Compression Booster

As seen in Sect. 2.3, compression boosting is one of the main applications of the PPC-paradigm. In particular, by Theorem 1, the problem of boosting a zero-order compressor is reducible to that of optimally partitioning the Burrows-Wheeler Transform. Ferragina et al. [12] proposed an efficient algorithm (hereafter denoted by LC_{OPT}) that, given an input string T, finds a partition of the $\texttt{BWT}(T)$ which is optimal among those ones induced by a leaf cover of the suffix tree of T (see Sect. 2.3). This partition can be computed in $O(n\sigma)$ time and its compression cost can be bounded in terms of effective k-th order entropy bounds (see Theorem 2). However, this is not the best possible partition for $\texttt{BWT}(T)$ given the 0-th order compressor to be boosted!

Consider an alphabet $\Sigma = \{c_1, c_2, \ldots, c_\sigma\}$ and assume that $c_1 < c_2 < \ldots < c_\sigma$. We divide Σ into $l = \sigma/\alpha$ sub-alphabets Σ_i of α consecutive symbols each, where $\alpha > 0$ will be defined later. For each Σ_i, we build a De Bruijn sequence T_i in which each pair of symbols of Σ_i but one[9] occurs exactly once. By construction, each sequence T_i has length α^2. Then, we construct the text $T = T_1 T_2 \ldots T_l$, so that $|T| = \sigma\alpha$ and each symbol of Σ occurs exactly α times in T.

[9]The reason for this exception is given by the fact that the sequence we are considering is not cyclic.

Let us now build $\texttt{BWT}(T)$ and consider its entire BW-matrix. Its first column is equal to $(c_1)^\alpha (c_2)^\alpha \ldots (c_\sigma)^\alpha$, whereas the last column $L = \texttt{BWT}(T)$ can be partitioned into substrings L_c each corresponding to the symbols of L whose row starts with c. By construction, L_c consists of α distinct symbols and the longest common prefix between any two suffixes of T is at most 1. Moreover, the concatenation of L_cs strings corresponding to symbols in the same sub-alphabet is formed by at most $\alpha + 1$ distinct symbols (namely, all the symbols are in the same sub-alphabet except one which belongs to an other sub-alphabet).

Since LC_{OPT} can partition only using prefix-close contexts (see [12]), there are just three possible partitions: (1) the one consisting of the whole L, (2) the one consisting of the σ substrings L_c, or (3) the one consisting of as many substrings as symbols of L. In order to compute the cost of each possible partition for LC_{OPT}, we have to fix the cost of storing the various models used by C. This cost clearly depends on the particular compressor in use. In the following we use the realistic assumption that each model is stored by paying $\log \sigma$ bits for each distinct symbol. This assumption covers the common case in which the compressor is either Huffman or Arithmetic.

1. Compressing the whole L at once costs at least $\sigma\alpha \log \sigma + \sigma \log \sigma = \Omega(\sigma\alpha \log \sigma)$ bits. In fact, all the symbols of Σ have the same frequency in L.
2. Compressing each string L_c costs at least $|L_c| \log |L_c| + |L_c| \log \sigma = \Omega(\alpha \log \sigma)$ bits, since each L_c contains α distinct symbols that occur once (and thus are equiprobable). The overall cost for this partition is $\Omega(\sigma\alpha \log \sigma)$ bits.
3. Compressing each symbol separately has overall cost at least $|L| \log \sigma = \Omega(\sigma\alpha \log \sigma)$ bits.

Therefore the best compression achieved by LC_{OPT} has cost $\Omega(\sigma\alpha \log \sigma)$ bits. Let us now consider another partition which is not prefix-close and thus it is not achievable by LC_{OPT}. This partition subdivides L into σ/α substrings denoted $S_1, S_2, \ldots, S_{\sigma/\alpha}$ of size α^2 symbols each (recall that $|T| = \sigma\alpha$). Notice that each S_i is drawn from an alphabet of size smaller than $\alpha + 1$, because it spans α L_c's that refer to symbols $c \in \Sigma_i$, with the addition of the single symbol that precedes T_i in T. The strings S_i are compressed individually in $O(\alpha^2 \log \alpha + \alpha \log \sigma)$ bits. Since there are σ/α strings S_is, the cost of this partition is $C = O(\sigma\alpha \log \alpha + \sigma \log \sigma)$. Therefore, by setting $\alpha = \Theta(\log \sigma)$, we have that $C = O(\sigma \log \sigma \log\log \sigma)$ bits.

Comparing the cost of LC_{OPT}'s partition with this one we observe that the former is $\Omega(\log \sigma / \log\log \sigma)$ times larger than the latter. Since $n = |T| = \sigma\alpha = \Theta(\sigma \log \sigma)$, we have that $\log \sigma / \log\log \sigma = \Theta(\log n / \log\log n)$.

As a result, no algorithm is currently known that is able to *boost* a zero-th order compressor by guaranteeing efficient bounds in terms of the optimal performance. We conclude this section by observing that Theorem 6 can be applied inside the boosting scheme of Sect. 2.3 to design a $(1 + \varepsilon)$-approximately optimal partition of the BWT with respect to a zero-order base compressor. This algorithm is also faster than the LC_{OPT} when the alphabet is larger than $\Omega(\text{polylog}(n))$; this case is interesting whenever we consider either a word-based BWT [28] or the XBW-transform for labeled trees [12].

9 Conclusion

In this paper we have investigated the problem of partitioning an input string T in such a way that compressing individually its parts via a base-compressor C gets a compressed output that is shorter than applying C over the entire T at once. We have provided the first algorithm which is guaranteed to compute in $O(n \log_{1+\varepsilon} n)$ time and $O(n)$ space a partition of T whose compressed output is guaranteed to be no more than $(1+\varepsilon)$-worse the optimal one, where ε may be any positive constant. This result has been extended to BWT-based compressors and to the compression booster of [12].

We point out that our results can be easily extended to *adaptive* compressors too, namely ones which compute the symbols' frequencies *incrementally* during the compression of the input text (see e.g. [21]). The typical example is adaptive Arithmetic coding and ppm (see [33]). In these cases, the concept of *adaptive empirical* entropy of T must be introduced to bound their performance. We will not detail our results for this setting, but just mention that hold the same time/space bounds stated in Theorems 6 and 7.

We point out that all our results are obtained by using a novel technique that permits to efficiently approximate solutions of Dynamic Programming recurrences in which the cost function satisfies a monotonicity requirement. We believe that our approximation scheme may find applications over a larger class of dynamic programming problems.

As future directions of research we suggest either to investigate the design of $o(n^2)$ algorithms for computing the *exact* optimal partition, and/or experiment and engineer our solution for the Web-applications considered in [10].

Acknowledgements We thank Raffaele Giancarlo for pointing out the terminology about the *PPC-paradigm* and for showing that our algorithmic solution to the text partitioning problem could be used as a tool for approximating efficiently the interesting class of Dynamic-Programming Recurrences we have dealt with in Sect. 3.

References

1. Bentley, J.L., McIlroy, M.D.: Data compression with long repeated strings. Inf. Sci. **135**(1–2), 1–11 (2001)
2. Bentley, J.L., Sleator, D.D., Tarjan, R.E., Wei, V.K.: A locally adaptive data compression scheme. Commun. ACM **29**(4), 320–330 (1986)
3. Buchsbaum, A.L., Giancarlo, R.: Table compression. In: Kao, M.Y. (ed.) Encyclopedia of Algorithms, pp. 939–942. Springer, Berlin (2008)
4. Buchsbaum, A.L., Caldwell, D.F., Church, K.W., Fowler, G.S., Muthukrishnan, S.: Engineering the compression of massive tables: an experimental approach. In: Proc. of 11th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 175–184 (2000)
5. Buchsbaum, A.L., Fowler, G.S., Giancarlo, R.: Improving table compression with combinatorial optimization. J. ACM **50**(6), 825–851 (2003)
6. Burrows, M., Wheeler, D.: A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation (1994)
7. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. ACM Trans. Comput. Syst. **26**(2) (2008)

8. Dasgupta, S., Papadimitriou, C., Vazirani, U.: Algorithms. McGraw-Hill Science/Engineering/Math, New York (2006)
9. Fenwick, P.M.: The burrows-wheeler transform for block sorting text compression: principles and improvements. Comput. J. **39**(9), 731–740 (1996)
10. Ferragina, P., Manzini, G.: On compressing the textual web. In: Proc. of Third ACM Conference on Web Search and Data Mining (WSDM) (2010)
11. Ferragina, P., Venturini, R.: The compressed permuterm index. ACM Trans. Algorithms (2010, to appear)
12. Ferragina, P., Giancarlo, R., Manzini, G., Sciortino, M.: Boosting textual compression in optimal linear time. J. ACM **52**, 688–713 (2005)
13. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Structuring labeled trees for optimal succinctness, and beyond. In: Proc. of 46th IEEE Symposium on Foundations of Computer Science (FOCS), pp. 184–193 (2005)
14. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Compressing and searching XML data via two zips. In: Proc. of 15th International World Wide Web Conference (WWW), pp. 751–760 (2006)
15. Ferragina, P., Giancarlo, R., Manzini, G.: The engineering of a compression boosting library: theory vs practice in BWT compression. In: Proc. of 14th European Symposium on Algorithms (ESA'06). LNCS, vol. 4168, pp. 756–767. Springer, Berlin (2006)
16. Ferragina, P., Giancarlo, R., Manzini, G.: The myriad virtues of wavelet trees. Inf. Comput. **207**, 849–866 (2009)
17. Giancarlo, R.: Dynamic programming: special cases. In: Apostolico, A., Galil, Z. (eds.) Pattern Matching Algorithms, 2nd edn., pp. 201–236. Oxford University Press, London (1997)
18. Giancarlo, R., Sciortino, M.: Optimal partitions of strings: a new class of Burrows-Wheeler compression algorithms. In: Proc. of 14th Symposium on Combinatorial Pattern Matching (CPM). LNCS, vol. 2676, pp.129–143. Springer, Berlin (2003)
19. Giancarlo, R., Restivo, A., Sciortino, M.: From first principles to the Burrows and Wheeler transform and beyond, via combinatorial optimization. Theor. Comput. Sci. **387**(3), 236–248 (2007)
20. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. of 14th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 841–850 (2003)
21. Howard, P.G., Vitter, J.S.: Analysis of arithmetic coding for data compression. Inf. Process. Manag. **28**(6), 749–764 (1992)
22. Kaplan, H., Landau, S., Verbin, E.: A simpler analysis of Burrows-Wheeler-based compression. Theor. Comput. Sci. **387**(3), 220–235 (2007)
23. Kosaraju, R., Manzini, G.: Compression of low entropy strings with Lempel–Ziv algorithms. SIAM J. Comput. **29**(3), 893–911 (1999)
24. Kulkarni, P., Douglis, F., LaVoie, J.D., Tracey, J.M.: Redundancy elimination within large collections of files. In: USENIX Annual Technical Conference, pp. 59–72 (2004)
25. Mäkinen, V., Navarro, G.: Position-restricted substring searching. In: Proc. of 7th Latin American Symposium on Theoretical Informatics (LATIN). LNCS, vol. 3887, pp. 703–714. Springer, Berlin (2006)
26. Mäkinen, V., Navarro, G.: Implicit compression boosting with applications to self-indexing. In: Proc. of 14th Symp. on String Processing and Information Retrieval (SPIRE). LNCS, vol. 4726, pp. 229–241. Springer, Berlin (2007)
27. Manzini, G.: An analysis of the Burrows-Wheeler transform. J. ACM **48**(3), 407–430 (2001)
28. Moffat, A., Isal, R.Y.: Word-based text compression using the Burrows-Wheeler transform. Inf. Process. Manag. **41**(5), 1175–1192 (2005)
29. Ouyang, Z., Memon, N.D., Suel, T., Trendafilov, D.: Cluster-based delta compression of a collection of files. In: Proc. of 3rd Conference on Web Information Systems Engineering (WISE), pp. 257–268. IEEE Comput. Soc., Los Alamitos (2002)
30. Suel, T., Memon, N.: Algorithms for delta compression and remote file synchronization. In: Sayood, K. (ed.) Lossless Compression Handbook. Academic Press, New York (2002)
31. Trendafilov, D., Memon, N., Suel, T.: Compressing file collections with a TSP-based approach. Technical Report TR-CIS-2004-02, Polytechnic University (2004)
32. Vo, B.D., Vo, K.-P.: Compressing table data with column dependency. Theor. Comput. Sci. **387**(3), 273–283 (2007)
33. Witten, I.H., Moffat, A., Bell, T.C.: Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd edn. Morgan Kaufmann, Los Altos (1999)