On optimally partitioning a text to improve its compression*

Paolo Ferragina Igor Nitto Rossano Venturini †

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 $\mathcal{C}$ gets a compressed output that is shorter than applying $\mathcal{C}$ over the entire T at once. This problem was introduced in [2, 3] in the context of table compression, and then further elaborated and extended to strings and trees by [10, 11, 21]. Unfortunately, the literature offers poor solutions: namely, we know either a cubic-time algorithm for computing the optimal partition based on dynamic programming [3, 15], or few heuristics that do not guarantee any bounds on the efficacy of their computed partition [2, 3], or algorithms that are efficient but work in some specific scenarios (such as the Burrows-Wheeler Transform, see e.g. [10, 21]) and achieve compression performance that might be worse than the optimal-partitioning by a $\Omega(\sqrt{\log n})$ factor. Therefore, computing efficiently the optimal solution is still open [4]. In this paper we provide the first algorithm which is guaranteed to compute in $O(n \log_{1+\epsilon} n)$ time a partition of T whose compressed output is guaranteed to be no more than $(1+\epsilon)$-worse the optimal one, where ϵ may be any positive constant.

1 Introduction

Reorganizing data in order to improve the performance of a given compressor $\mathcal{C}$ is a recent and important paradigm in data compression (see e.g. [3, 10]). The basic idea consist of *permuting* the input data T to form a new string T' which is then *partitioned* into substrings $T' = T'_1 T'_2 \cdots T'_k$ that are finally compressed *individually* by the base compressor $\mathcal{C}$. 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 $\mathcal{C}$ to the entire T: just take the identity permutation and set $k = 1$. 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

*It has been partially supported by Yahoo! Research, Italian MIUR Italy-Israel FIRB Project. The authors' address is Dipartimento di Informatica, L.go B. Pontecorvo 3, 56127 Pisa, Italy.

†Department of Computer Science, University of Pisa. emails: {ferragina, nitto, rventurini}@di.unipi.it

weak compressor $\mathcal{C}$. 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 [10], trees [11], tables [3], etc.), different granularities for the items of T to be permuted (chars, node labels, columns, blocks [1, 19], files [6, 25, 26], etc.), different permutations (see e.g. [14, 27, 26, 6]), 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 [2], and further elaborated upon in [3]. 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 [3] 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 (see below), or some very simple and fast algorithms which however did not have any guaranteed bounds in terms of efficacy of their grouping process. 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). Furthermore, they showed that the combination of the TSP-heuristic with the DP-optimal partitioning is even better, but it is too slow to be used in practice even on small file sizes 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 [5] (shortly `BWT`) with a context-based grouping of the input characters, which are finally compressed via proper 0-th order-entropy compressors (like `MTF`, `RLE`, Huffman, Arithmetic, or their combinations, see e.g. [28]). Here the PPC-paradigm takes the name of *compression booster* [10] because the net result it produces is to boost the performance of the base compressor $\mathcal{C}$ from 0-th order-entropy bounds to k-th order entropy bounds, simultaneously over all $k \geq 0$. In this scenario the permutation acts on single characters, and the partitioning/permuting steps deploy the context (substring) following each symbol in the original string in order to identify "related" characters which must be therefore compressed together. Recently [14] investigated whether do exist other permutations of the characters 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 [22, 9, 17, 21] and have shown, also experimentally [8], that the partitioning of the BW-transformed data is a key step for achieving effective compression ratios. Optimal partitioning is actually

[1] Page 836 of [3] 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 DP."

even more mandatory in the context of labeled-tree compression where a BWT-inspired transform, called XBW-transform in [11, 12], allows to produce permuted strings with a strong clustering effect. Starting from these premises [15] attacked the computation of the optimal partitioning of T via a DP-approach, which turned to be very costly; then [10] (and subsequently many other authors, see e.g. [9, 21, 11]) 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 Appendix A in which we show an infinite class of strings for which the compression achieved by the booster is far from the optimal-partitioning by a multiplicative factor $\Omega(\sqrt{\log n})$.

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 as `gzip`, `bzip2`, `ppm`, etc. [28]. 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` [28] and the family of `BWT`-based compressors [8], 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 $\mathcal{C}$ 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 $T = (2^k 0)^{n/(2(k+1))}(2^k 1)^{n/(2(k+1))}$ and observe that if we divide T in two halves and compress them individually, the output size is about $O(\log n)$ bits, but if we compress the entire T at once 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 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 [25, 6, 24, 26]. In these cases, the partitioning step looks for *homogeneous* groups of contiguous files which can be effectively compressed together by the base-compressor $\mathcal{C}$. More than before, taking the largest memory-footprint offered by $\mathcal{C}$ 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). Again, in all those cases we could apply the optimal DP-based partitioning approach of [15, 3], but this would take more than cubic time (in the overall input size $|T|$) thus resulting unusable even on small input data of few MBs!

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 [4]). 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 $\mathcal{C}$ 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 $\mathcal{C}$. So, we are assuming that T has been (possibly) permuted in advance, and we are concentrating on the last two steps of the PPC-paradigm. Now, given a partition $\mathcal{P}$ of the input string into contiguous substrings, say $T = T_1 T_2 \cdots T_k$, we denote by $\texttt{Cost}(\mathcal{P})$ the cost of this partition and measure it as $\sum_{i=1}^{l} |\mathcal{C}(T_i)|$, where $|\mathcal{C}(\alpha)|$ is the length in bit of the string α compressed by $\mathcal{C}$. The problem of *optimally partitioning* T according to the base-compressor $\mathcal{C}$ consists then of computing the partition $\mathcal{P}_{\texttt{opt}}$ achieving the minimum cost, namely $\mathcal{P}_{\texttt{opt}} = \min_{\mathcal{P}} \texttt{Cost}(\mathcal{P})$, and thus the shortest compressed output.[2]

As we mentioned above $\mathcal{P}_{\texttt{opt}}$ might be computed via a Dynamic-Programming approach [3, 15]. Define $E[i]$ as the cost of the optimum partitioning of $T[1, i]$, and set $E[0] = 0$. Then, for each $i \geq 1$, we can compute $E[i]$ as the $min_{0 \leq j \leq i-1} E[j] + |\mathcal{C}(T[j+1, i])|$. At the end $E[n]$ gives the cost of $\mathcal{P}_{\texttt{opt}}$, which can be explicitly determined by standard back-tracking over the DP-array. Unfortunately, this solution requires to run $\mathcal{C}$ 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 $\mathcal{C}$ over each substring of T, we use an entropy-based estimation of $\mathcal{C}$'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 $(1 + \epsilon)$ worse than $\mathcal{P}_{\texttt{opt}}$, where ϵ may be any positive constant. Item (1) takes inspiration from the heuristics proposed in [2, 3], 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 [2, 3]). Item (2) boils down to show that the optimal partitioning problem can be rephrased as a Single Source Shortest path computation over a weighted DAG consisting of n nodes and $O(n^2)$ edges whose costs are derived from item (1). We prove some interesting structural properties of this graph that allow us to restrict the computation of that SSSP to a subgraph consisting of $O(n \log_{1+\epsilon} n)$ edges only. The technical part of this paper (see Section 3) 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 $(1 + \epsilon)$-approximate $\mathcal{P}_{\texttt{opt}}$ in $O(n \log_{1+\epsilon} n)$ time and $O(n)$ space, for both 0-th order compressors (like Huffman and Arithmetic [28]) and k-th order compressors (like ppm [28]).

[2] We are assuming that $\mathcal{C}(\alpha)$ 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.

We will also extend these results to the class of BWT-based compressors, when T is a collection of texts.

We point out that the result on 0-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 [28], and from the theoretical side since it can be applied to the compression booster of [10] to fast obtain an approximation of the optimal partition of $\mathtt{BWT}(T)$ in $O(n \log_{1+\epsilon} n)$ time. This may be better than the algorithm of [10] both in time complexity, since that takes $O(n|\Sigma|)$ time where Σ is the alphabet of T, and in compression ratio (as we have shown above, see Appendix A). The case of a large alphabet (namely, $|\Sigma| = \Omega(\mathtt{polylog}(n))$) is particularly interesting whenever we consider either a word-based BWT [23] or the XBW-transform over labeled trees [10]. Finally, we mention that our results apply also to the practical case in which the base compressor $\mathcal{C}$ 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 reduces to $O(n \log_{1+\epsilon}(B \log \sigma))$.

The map of the paper is as follows. Section 2 introduces some basic notation and terminology. Section 3 describes our reduction from the optimal partitioning problem of T to a SSSP problem over a weighted DAG in which edges represent substrings of T and edge costs are entropy-based estimations of the compression of these substrings via $\mathcal{C}$. The subsequent Sections will address the problem of incrementally and efficiently computing those edge costs as they are needed by the SSSP-computation, distinguishing the two cases of 0-th order estimators (Section 4) and k-th order estimators (Section 5), and the situation in which $\mathcal{C}$ is a BWT-based compressor and T is a collection of files (Section 6).

2 Notation

In this paper we will use entropy-based upper bounds for the estimation of $|\mathcal{C}(T[i,j])|$, so we need to recall some basic notation and terminology about entropies. Let $T[1,n]$ be a string 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 $H_0(T) = \frac{1}{|T|} \sum_{c \in \Sigma}^{h} n_c \log \frac{n}{n_c}$.

Recall that $|T|H_0(T)$ provides an information-theoretic lower bound to the output size of any compressor that encodes each symbol of T with a fixed code [28]. The so-called zero-th order statistical compressors (such as Huffman or Arithmetic [28]) 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 scanning the input text (*semistatic* model). 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 $|\mathcal{C}_0(T)| \leq \lambda n H_0(T) + f_0(n, \sigma)$ bits, where λ is a positive constant and $f_0(n, \sigma)$ is a function including the extra costs of encoding the source model and/or other inefficiencies of $\mathcal{C}$. In the following we will assume that the function $f_0(n, \sigma)$ can be computed in constant time given n and σ. As an example, for Huffman $f_0(n, \sigma) = \sigma \log \sigma + n$ bits and $\lambda = 1$, and for Arithmetic $f_0(n, \sigma) = \sigma \log n + \log n/n$ bits and $\lambda = 1$.

In order to evict the cost of the model, we can resort to zero-th order *adaptive* compressors that do not require to know the symbols' frequencies in advance, since they are computed incrementally during the compression. The zero-th order *adaptive empirical* entropy of T [16] is then defined as $H_0^a(T) = \frac{1}{|T|}\sum_{c\in\Sigma}^{h} \log \frac{n!}{n_c!}$ We will bound the compress size achieved by zero-th order adaptive compressors over T by $|\mathcal{C}_0^a(T)| = nH_0^a(T)$ bits.

Let us now come to more powerful compressors. For any string 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 $H_k(T) = \frac{1}{|T|}\sum_{u\in\Sigma^k} |u_T| H_0(u_T)$. Analogously, the k-th order *adaptive empirical* entropy of T is defined as $H_k^a(T) = \frac{1}{|T|}\sum_{u\in\Sigma^k} |u_T|\, H_0^a(u_T)$

We have $H_k(T) \geq H_{k+1}(T)$ for any $k \geq 0$. As usual in data compression [22], the value $nH_k(T)$ is an information-theoretic lower bound to the output size of any compressor that encodes each symbol of T with a fixed code that depends on the symbol itself and on the k immediately preceding symbols. Recently (see e.g. [18, 22, 10, 9, 21, 11] and refs therein) authors have provided *upper bounds* in terms of $H_k(|T|)$ for sophisticated data-compression algorithms, such as `gzip` [18], `bzip2` [22, 10, 17], and `ppm`. These bounds have the form $|\mathcal{C}(T)| \leq \lambda |T|\, H_k(T) + f_k(|T|, \sigma)$, 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 $\mathcal{C}$. The smaller are λ and $f_k()$, the better is the compressor $\mathcal{C}$. As an example, the bound of the compressor in [21] has $\lambda = 1$ and $f(|T|, \sigma) = O(\sigma^{k+1} \log |T| + |T| \log \sigma \log\log |T| / \log |T|)$. Similar bounds that involve the adaptive k-th order entropy are known [22, 10, 9] for many compressors. In these cases the bound takes the form $|\mathcal{C}_k^a(T)| = \lambda |T| H_k^*(T) + g_k(\sigma)$ bits, where the value of g_k depends only on the alphabet size σ.

In our paper we will use these entropy-based bounds for the estimation of $|\mathcal{C}(T[i,j])|$, but of course this will not be enough to achieve a fast DP-based algorithm 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 (Section 3) and introduce few novel technicalities (Sections 4–5) that will allow us to compute $H_k(T[i,j])$ only on a *reduced* subset of T's substrings, having size $O(n \log_{1+\epsilon} n)$, by taking $O(\texttt{polylog}(n))$ time per substring and $O(n)$ space overall.

3 The problem and our solution

The optimal partitioning problem, stated in Section 1 can be reduced to a single source shortest path computation (SSSP) over a directed acyclic graph $\mathcal{G}(T)$ defined as follows. The graph $\mathcal{G}(T)$ 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 for any pair of indices i and j such that $i < j$. Each edge (v_i, v_j) has associated the cost $c(v_i, v_j) = |\mathcal{C}(T[i, j-1])|$ that corresponds to the size in bits of the substring $T[i, j-1]$ compressed by $\mathcal{C}$. We remark the following crucial, but easy to prove, property of the cost function defined on $\mathcal{G}(T)$:

Fact 1 *For any vertex v_i, it is* $0 < c(v_i, v_{i+1}) \leq c(v_i, v_{i+2}) \leq \ldots \leq c(v_i, v_{n+1})$

There is a one-to-one correspondence between paths from v_1 to v_{n+1} in $\mathcal{G}(T)$ and partitions of T: every edge (v_i, v_j) in the path identifies a contiguous substring $T[i, j-1]$ of the corresponding partition. Therefore the cost of a path is equal to the (compression-)cost of the corresponding partition. Thus, we can find the optimal partition of T by computing the shortest path in $\mathcal{G}(T)$ from v_1 to v_{n+1}. Unfortunately this simple approach has two main drawbacks:

1. the number of edges in $\mathcal{G}(T)$ is $\Theta(n^2)$, thus making the SSSP computation inefficient (i.e. $\Omega(n^2)$ time) if executed directly over $\mathcal{G}(T)$;
2. the computation of the each edge cost might take $\Theta(n)$ time over most T's substrings, if $\mathcal{C}$ is run on each of them from scratch.

In the following sections we will successfully address both these two drawbacks. First, we sensibly reduce the number of edges in the graph $\mathcal{G}(T)$ to be examined during the SSSP computation and show that we can obtain a $(1+\epsilon)$ approximation using only $O(n \log_{1+\epsilon} n)$ edges, where $\epsilon > 0$ is a user-defined parameter (Section 3.1). Second, we show some sufficient properties that $\mathcal{C}$ needs to satisfy in order to compute efficiently every edge's cost. These properties hold for some well-known compressors— e.g. 0-order compressors, `PPM`-like and `bzip`-like compressors— and for them we show how to compute each edge cost in constant or polylogarithmic time (Sections 4—6).

3.1 A pruning strategy

The aim of this section is to design a *pruning* strategy that produces a subgraph $\mathcal{G}_\epsilon(T)$ of the original DAG $\mathcal{G}(T)$ in which the shortest path distance between its leftmost and rightmost nodes, v_1 and v_{n+1}, increases by no more than a factor $(1+\epsilon)$. We define $\mathcal{G}_\epsilon(T)$ to contain all edges (v_i, v_j) of $\mathcal{G}(T)$, recall $i < j$, such that at least one of the following two conditions holds:

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

In other words, by fact 1, we are keeping for each integer k the edge of $\mathcal{G}(T)$ that approximates at the best the value $(1+\epsilon)^k$ from below. Given this, we will call *ϵ-maximal* the edges of $\mathcal{G}_\epsilon(T)$. Clearly, each vertex of $\mathcal{G}_\epsilon(T)$ has at most $\log_{1+\epsilon} n = O(\frac{1}{\epsilon} \log n)$ outgoing edges, which are ϵ-maximal by definition. Therefore the total size of $\mathcal{G}_\epsilon(T)$ is at most $O(\frac{n}{\epsilon} \log n)$. Hereafter, we will denote with $d_G(-,-)$ the shortest path distance between any two nodes in a graph G.

The following lemma states a basic property of shortest path distances over our special DAG $\mathcal{G}(T)$:

Lemma 1 *For any triple of indices $1 \leq i \leq j \leq q \leq n+1$ we have:*

1. $d_{\mathcal{G}(T)}(v_j, v_q) \leq d_{\mathcal{G}(T)}(v_i, v_q)$
2. $d_{\mathcal{G}(T)}(v_i, v_j) \leq d_{\mathcal{G}(T)}(v_i, v_q)$

Proof: We prove just 1, since 2 is symmetric. It suffices by induction to prove the case $j = i+1$. Let $(v_i, w_1)(w_1, w_2)...(w_{h-1}, w_h)$, with $w_h = v_q$, be a shortest path in $\mathcal{G}(T)$ from v_i to v_q. By fact 1, $c(v_j, w_1) \leq c(v_i, w_1)$ since $i \leq j$. Therefore the cost of the path $(v_j, w_1)(w_1, w_2)...(w_{h-1}, w_h)$ is at most $d_{\mathcal{G}(T)}(v_i, v_q)$, which proves the claim. ∎

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

Theorem 1 *For any text T, the shortest path in $\mathcal{G}_\epsilon(T)$ from v_1 to v_{n+1} has a total cost of at most $(1+\epsilon)\ d_{\mathcal{G}(T)}(v_1, v_{n+1})$.*

Proof: We prove a stronger assertion: $d_{\mathcal{G}_\epsilon(T)}(v_i, v_{n+1}) \leq (1+\epsilon)\ d_{\mathcal{G}(T)}(v_i, v_{n+1})$ for any index $1 \leq i \leq n+1$. This is clearly true for $i = n+1$, because in that case the distance is 0. Now let us inductively consider the shortest path π in $\mathcal{G}(T)$ from v_i to v_{n+1} and let $(v_k, v_{t_1})(v_{t_1}, v_{t_2}) \ldots (v_{t_h} v_{n+1})$ be its edges. By the definition of ϵ-maximal edge, it is possible to find an ϵ-maximal edge (v_k, v_r) with $t_1 \leq r$, such that $c(v_k, v_r) \leq (1+\epsilon)\ c(v_k, v_{t_1})$. By Lemma 1, $d_{\mathcal{G}(T)}(v_r, v_{n+1}) \leq d_{\mathcal{G}(T)}(v_{t_1}, v_{n+1})$. By induction, $d_{\mathcal{G}_\epsilon(T)}(v_r, v_{n+1}) \leq (1+\epsilon)\ d_{\mathcal{G}(T)}(v_r, v_{n+1})$. Combining this with the triangle inequality we get the thesis. ∎

3.2 Space and time efficient algorithms for generating $\mathcal{G}_\epsilon(T)$

Theorem 1 ensures that, in order to compute a $(1+\epsilon)$ approximation of the optimal partition of T, it suffices to compute the SSSP in $\mathcal{G}_\epsilon(T)$ from v_1 to v_{n+1}. This can be easily computed in $O(|\mathcal{G}_\epsilon(T)|) = O(n \log_\epsilon n)$ time since $\mathcal{G}_\epsilon(T)$ is a DAG [7], by making a single pass over its vertices and relaxing all edges going out from the current one.

However, generating $\mathcal{G}_\epsilon(T)$ in efficient time is a non-trivial task for three main reasons. First, the original graph $\mathcal{G}(T)$ contains $\Omega(n^2)$ edges, so that we cannot check each of them to determine whether it is ϵ-maximal or not, because this would take $\Omega(n^2)$ time. Second, we cannot compute the cost of an edge (v_i, v_j) by executing $\mathcal{C}(T[i, j-1])$ *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 $\mathcal{G}_\epsilon(T)$ (e.g. its adjacency lists) because it consists of $\Theta(n\ \texttt{polylog}(n))$ edges, and thus its space occupancy would be super-linear in the input size.

The rest of this section is devoted to design an algorithm which overcomes the three limitations above. The specialty of our algorithm consists of materializing $\mathcal{G}_\epsilon(T)$ on-the-fly, as its vertices are examined during the SSSP-computation, by spending only polylogarithmic time per edge. The actual time complexity per edge will depend on the entropy-based cost function we will use to estimate $|\mathcal{C}(T[i, j-1])|$ (see Section 2) and on the dynamic data structure we will deploy to compute that estimation efficiently.

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 design a general framework for solving our problem; in the next sections we will then provide implementations of this data structure and thus obtain real time/space bounds for our problem. So, let us assume to have a dynamic data structure that maintains a set of *sliding windows* over T denoted by $w_1, w_2, \ldots, w_{\log_{1+\epsilon} n}$.

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 $r_1 \leq r_2 \leq \ldots \leq r_{\log_{1+\epsilon} n}$. The data structure must support the following three operations:

1. $\texttt{Remove}()$ moves the starting position l of all windows one position to the right (i.e. $l+1$);
2. $\texttt{Append}(w_i)$ moves the ending position of the window w_i one position to the right (i.e. $r_i + 1$);
3. $\texttt{Size}(w_i)$ computes and returns the value $|\mathcal{C}(T[l, r_i])|$.

This data structure is enough to generate ϵ-maximal edges via a single pass over T, using $O(|T|)$ 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 satisfying $c(v_l, v_{1+r_t}) \leq (1+\epsilon)^t < c(v_l, v_{1+(r_t+1)})$. Initially all indices are set to 0. To maintain the invariant, when the text scan advances to the next position $l+1$, we call operation $\texttt{Remove}()$ once to increment index l and, for each $t = 1, \ldots, \log_{1+\epsilon}(n)$, we call operation $\texttt{Append}(w_t)$ until we find the largest r_t such that $\texttt{Size}(w_t) = c(v_l, v_{1+r_t}) \leq (1+\epsilon)^t$. The key issue here is that $\texttt{Append}$ and $\texttt{Size}$ are paired so that our data structure should take advantage of the rightward sliding of r_t for computing $c(v_l, v_{1+r_t})$ efficiently. Just one character is entering w_t to its right, so we need to deploy this fact for making the computation of $\texttt{Size}(w_t)$ 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 Section 2, to estimate indeed the edge costs $\texttt{Size}(w_t) = |\mathcal{C}(w_t)|$. 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 $\texttt{Append} + \texttt{Size}$. These issues will be discussed in the following sections, here we just state that, overall, the SSSP computation over $\mathcal{G}_\epsilon(T)$ takes $O(n)$ calls to operation $\texttt{Remove}$, and $O(n \log_{1+\epsilon} n)$ calls to operations $\texttt{Append}$ and $\texttt{Size}$.

Theorem 2 *If we have a dynamic data structure occupying $O(n)$ space and supporting operation* $\texttt{Remove}$ *in time $L(n)$, and operations* $\texttt{Append}$ *and* $\texttt{Size}$ *in time $R(n)$, then we can compute the shortest path in $\mathcal{G}_\epsilon(T)$ from v_1 to v_{n+1} taking $O(n\, L(n) + (n \log_{1+\epsilon} n)\, R(n))$ time and $O(n)$ space.*

4 On zero-th order compressors

In this section we explain how to implement the data structure above whenever $\mathcal{C}$ is a 0-th order compressor, and thus H_0 is used to provide a bound to the compression cost of $\mathcal{G}(T)$'s edges (see Section 2). The key point is actually to show how to efficiently compute $\texttt{Size}(w_i)$ as the sum of $|T[l, r_i]| H_0(T[l, r_i]) = \sum_{c \in \Sigma} n_c \log((r_i - l + 1)/n_c)$ (see its definition in Section 2) 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 $T[l, r_i]$.

The first solution we are going to present is very simple and uses $O(\sigma)$ space per window. 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 of $\sum_{c \in \Sigma} A_i[c] \log A_i[c]$. It is easy to notice that:

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

Therefore, if we know the value of E_i, we can answer to a query $\texttt{Size}(w_i)$ in constant time. So, we are left with showing how to implement efficiently the two operations that modify l or any rs value and, thus, modify appropriately the E's value. This can be done as follows:

1. $\texttt{Remove}()$: For each window w_i, we subtract from the appropriate counter and from 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 $A_i[T[r+1]]$ 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 $r_i = r_i + 1$.

In this way, operation $\texttt{Remove}$ requires constant time per window, hence $O(\log_{1+\epsilon} n)$ time overall. $\texttt{Append}(w_i)$ takes constant time. The space required by the counters A_i is $O(\sigma \log_{1+\epsilon} n)$ words. Unfortunately, the space complexity of this solution can be too much when it is used as the basic-block for computing the k-th order entropy of T (see Section 2) as we will do in Section 5. In fact, we would achieve $\min(\sigma^{k+1} \log_{1+\epsilon} n, n \log_{1+\epsilon} n)$ space, which may be superlinear in n depending on σ and k.

The rest of this section is therefore devoted to provide an implementation of our dynamic data structure that takes the same query time above for these three operations, but within $O(n)$ space, which is independent of σ and k. The new solution still uses E's value but the counters A_i are computed on-the-fly by exploiting the fact that all windows share the same value of l. We keep an array B indexed by symbols whose entry $B[c]$ stores the number of occurrences of c in $T[1, l]$. We can keep these counters updated after a $\texttt{Remove}$ by simply decreasing $B[T[l]]$ by one. We also maintain an array R with an entry for each text position. The 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 no more than n, hence they take $O(n)$ space.

These two arrays are enough to correctly update the value E_i after $\texttt{Append}(w_i)$, which is in turn enough to estimate H_0 (see Eqn 1). In fact, we can compute the value $A_i[T[r_i + 1]]$ by computing $R[r_i + 1] - B[T[r_i + 1]]$ which correctly reports the number of occurrences of $T[r_i + 1]$ in $T[l \ldots r_i + 1]$. Once we have the value of $A_i[T[r_i + 1]]$, we can update E_i as explained in the above item 2.

We are left with showing how to support $\texttt{Remove}()$ whose computation requires to evaluate the value of $A_i[T[l]]$ for each window w_i. Each of these values can be computed as $R[t] - B[T[l]]$ where t is the last occurrence of symbol $T[l]$ in $T[l, r_i]$. The problem here is given by the fact 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. In this case we force that position to occur in $L_{T[j]}$ just once. These lists are sufficient to compute values $A_i[T[l]]$ for all the windows together. In fact, since any position in $L_{T[l]}$ is the last occurrence of at least one sliding window, each of them can be used to compute $A_i[T[l]]$ for the appropriate indices i. Once we have all values $A_i[T[l]]$, we can update all E_i's as explained in the above item 1. Since list $L_{T[l]}$ contains no more than $\log_{1+\epsilon} n$ elements, all Es can be updated in $O(\log_{1+\epsilon} n)$ time. Notice that the number of elements in all the lists L is bounded by the text length. Thus, they are stored using $O(n)$ space.

It remains to explain how to keep lists L correctly updated. Notice that only one list may change after a $\texttt{Remove}()$ or an $\texttt{Append}(w_i)$. In the former case we have possibly to remove position l from list $L_{T[l]}$. This operation is simple because, if that position is in the list, then $T[l]$ 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) and, thus, it must be the head of $L_{T[l]}$. The case of $\texttt{Append}(w_i)$ 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. Recall that $\texttt{Append}(w_i)$ inserts symbol $T[r_i+1]$ in w_i. Thus, it must be inserted in $L_{T[r_i+1]}$ in its correct (sorted) position, if it is not present yet. Obviously, we can do that in $O(\log_{1+\epsilon} n)$ time by scanning the whole list. This is too much, so we show how to spend only constant time. Let p the rightmost occurrence of the symbol $T[r_i + 1]$ in $T[0, r_i]$.[3] 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 $w_{r_{i-1}}$). In any case, the list is kept updated in constant time.

The following Lemma derives by the discussion above:

Lemma 2 *Let $T[1, n]$ be a text drawn from an alphabet of size $\sigma = \texttt{poly}(n)$. If we estimate $\texttt{Size}()$ via 0-th order entropy (as detailed in Section 2), then we can design a dynamic data structure that takes $O(n)$ space and supports the operations $\texttt{Remove}$ in $R(n) = O(\log_{1+\epsilon} n)$ time, and $\texttt{Append}$ and $\texttt{Size}$ in $L(n) = O(1)$ time.*

In order to evict the cost of the model from the compressed output (see Section 2), authors typically resort to zero-th order *adaptive* compressors which do not store the symbols' frequencies, since they are computed *incrementally* during the compression [16]. A similar approach can be used in this case to achieve the same time and space bounds of Lemma 2. Here, we require that $\texttt{Size}(w_i) = |\mathcal{C}_0^a(T[l, r_i])| = |T[l, r_i]| H_0^a(T[l, r_i])$. Recall that with these type of compressors the model must not be stored. We use the same tools above but we

[3] 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.

change the values stored in variables E_i and the way in which they are updated after a `Remove` or an `Append`.

Observe that in this case we have that

$$|\mathcal{C}_0^a(T[l, r_i])| = |T[l, r_i]| H_0^a(T[l, r_i]) = \log((r_i - l + 1)!) - \sum_{c \in \Sigma} \log(n_c!)$$

where n_c is the number of occurrences of symbol c in $T[l, r_i]$. Therefore, if the variable E_i stores the value $\sum_{c\in\Sigma} \log(A_i[c]!)$, then we have that $|T[l, r_i]| H_0^a(T[l, r_i]) = \log((r_i - l + 1)!) - E_i$.[4]

After the two operations, we change E's value in the following way:

1. `Remove`(): For any window w_i we update E_i by subtracting $\log(A_i[T[l]])$. We also increase l by one.

2. `Append`(w_i): We update E_i by summing $\log A[T[r_i + 1]]$ and we increase r_i by one.

By the discussion above and Theorem 2 we obtain:

Theorem 3 *Given a text $T[1, n]$ drawn from an alphabet of size $\sigma = \mathtt{poly}(n)$, we can find an $(1 + \epsilon)$-optimal partition of T with respect to a 0-th order (adaptive) compressor in $O(n \log_{1+\epsilon} n)$ time and $O(n)$ space, where ϵ is any positive constant.*

We point out that these results can be applied to the compression booster of [10] to fast obtain an approximation of the optimal partition of `BWT`(T). This may be better than the algorithm of [10] both in time complexity, since that algorithm took $O(n\sigma)$ time, and in compression ratio by a factor up to $\Omega(\sqrt{\log n})$ (see the discussion in Section 1). The case of a large alphabet (namely, $\sigma = \Omega(\mathtt{polylog}(n))$) is particularly interesting whenever we consider either a word-based `BWT` [23] or the `XBW`-transform over labeled trees [10]. We notice that our result is interesting also for the *Huffword* compressor which is the standard choice for the storage of Web pages [28]; here Σ consists of the distinct words constituting the Web-page collection.

5 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 Section 2). Here `Size`(w_i) must compute $|\mathcal{C}(T[l, r_i])|$ which is estimated by $(r_i - l + 1)H_k(T[l, r_i]) + f_k(r_i - l + 1, |\Sigma_{T[l,r_i]}|)$, where $\Sigma_{T[l,r_i]}$ denotes the number of different symbols in $T[l, r_i]$..

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 $[1, n]$ without modifying its zero-th order entropy. In fact the number of distinct q-grams occurring in T_q is less than n, the length of T. 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.

[4]Notice that the value $\log((r_i - l + 1)!)$ can be stored in a variable and updated in constant time since the size of the value $r_i - l + 1$ changes just by one after a `Remove` or an `Append`.

A simple calculation shows that the k-th order (adaptive) entropy of a string (see definition Section 2) can be expressed as the difference between the zero-th order (adaptive) entropy of its $k+1$-grams and its k-grams. This suggests that we can use the solution of the previous section in order to compute 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 3 (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 3 *Let $T[1,n]$ be a text drawn from an alphabet of size $\sigma = \texttt{poly}(n)$. If we estimate* `Size()` *via k-th order entropy (as detailed in Section 2), then we can design a dynamic data structure that takes $O(n)$ space and supports the operations* `Remove` *in $R(n) = O(\log_{1+\epsilon} n)$ time, and* `Append` *and* `Size` *in $L(n) = O(1)$ time.*

Essentially the same technique is applicable to the case of k-th order *adaptive* compressor $\mathcal{C}$, in this case we keep up-to-date the 0-th order *adaptive* entropies of the strings T_{k+1} and T_k (details in [**?**]).

Theorem 4 *Given a text $T[1,n]$ drawn from an alphabet of size $\sigma = \texttt{poly}(n)$, we can find an $(1+\epsilon)$-optimal partition of T with respect to a k-th order (adaptive) compressor in $O(n \log_{1+\epsilon} n)$ 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 $\mathcal{C}$ has a maximum (block) size B of data it can process at once (this is the typical scenario for `gzip`, `bzip2`, etc.). In this situation the time performance of our solution reduces to $O(n \log_{1+\epsilon}(B \log \sigma))$.

6 On `BWT`-based compressors

As we mentioned in Section 2 we know entropy-bounded estimates for the output size of `BWT`-based compressors. So we could apply Theorem 4 to compute the optimal partitioning of T for such a type of compressors. Nevertheless, it is also known [8] that such compression-estimates are rough in practice because of the features of the compressors that are applied to the `BWT`(T)-string. Typically, `BWT` is encoded via a sequence of simple compressors such as `MTF` encoding, `RLE` encoding (which is optional), and finally a 0-order encoder like Huffman or Arithmetic [28]. For each of these compression steps, a 0-th entropy bound is known [22], but the combination of these bounds may result much far from the final compressed size produced by the overall sequence of compressors in practice [8].

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 4, but with the advantage of computing the $(1+\epsilon)$-optimal solution wrt the real compressed size, thus without any estimation by any entropy-cost functions. Since in practice it is $\sigma = \texttt{polylog}(n)$, this slowdown should be negligible. In order to achieve this result, we need to address a slightly different (but yet interesting in practice) problem which is defined as follows. 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 characters greater than any 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 cost (as defined in Section 1) is at most $(1+\epsilon)$ the minimum possible cost, for any fixed $\epsilon > 0$. We notice that this problem generalizes the table partitioning problem [3], since we can assume that $S[i]$ is a column of the table.

To simplify things we will drop the `RLE` encoding step of a `BWT`-based algorithm, and defer the complete solution to the full version of this paper. We start by noticing that a close analog of Theorem 2 holds for this variant of the optimal partitioning problem, which implies that a $(1+\epsilon)$-approximation of the optimum cost (and the corresponding partition) can be computed using a data structure supporting operations `Append`, `Remove`, and `Size`; with the only difference that the windows $w_1, w_2, \ldots, w_m$ subject to the operations are groups of contiguous pages of the form $w_i = S[l, r_i]$.

It goes without saying that there exist data structures designed to dynamically maintain a dynamic text compressed with a `BWT`-based compressor under insertions and deletions of symbols (see [13] and references therein). But they 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 overall space complexity.

Instead of keeping a given window w in compressed form, our approach will only store the frequency distribution of the integers in the string $w' = \mathtt{MTF}(\mathtt{BWT}(w))$ since this is enough to compute the compressed output size produced by the final step of the `BWT`-based algorithm, which is usually implemented via Huffman or Arithmetic [28]. Indeed, since `MTF` produces a sequence of integers from 0 to σ, we can store their number of occurrences for each window w_i into an array F_{w_i} of size σ. The update of F_{w_i} due to the insertion or the removal of a page in w_i incurs two main difficulties: (1) how to update w'_i as pages are added/removed from the extremes of the window w_i, (2) perform this update implicitly over F_{w_i}, because of the space reasons mentioned above. Our solution relies on two key facts about `BWT` and `MTF`:

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 (see [13]).
2. If a string s' is obtained from string s by inserting or removing a char c into an arbitrary position, then $\mathtt{MTF}(s')$ differs from $\mathtt{MTF}(s)$ in at most σ symbols. 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 first occurrence of each symbol of Σ among c and c'.

Due to space limitations we defer the solution to the Appendix B, and state here the result we are able to achieve.

Theorem 5 *Given a sequence of texts of total length n and alphabet size $\sigma = \mathtt{poly}(n)$, we can compute an $(1+\epsilon)$-approximate solution to the optimal partitioning problem for a* `BWT`*-based compressor, in $O(n(\log_{1+\epsilon} n)\ \sigma\ \log n)$ time and $O(n + \sigma \log_{1+\epsilon} n)$ space.*

7 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 $\mathcal{C}$ gets a compressed output that is shorter than applying $\mathcal{C}$ over the entire T at once. We provide the first algorithm which is guaranteed to compute in $O(n \log_{1+\epsilon} n)$ time a partition of T whose compressed output is guaranteed to be no more than $(1+\epsilon)$-worse the optimal one, where ϵ may be any positive constant. As future directions of research we would like either to investigate the design of $o(n^2)$ algorithms for computing the *exact* optimal partition, and/or experiment and engineer our solution over large datasets.

References

[1] J.L. Bentley and M.D. McIlroy. Data compression with long repeated strings. *Information Sciences*, 135(1-2):1–11, 2001.

[2] A. L. Buchsbaum, D. F. Caldwell, K. W. Church, G. S. Fowler, and S. Muthukrishnan. Engineering the compression of massive tables: an experimental approach. In *Procs ACM-SIAM SODA*, pages 175–184, 2000.

[3] Adam L. Buchsbaum, Glenn S. Fowler, and Raffaele Giancarlo. Improving table compression with combinatorial optimization. *J. ACM*, 50(6):825–851, 2003.

[4] A.L. Buchsbaum and R. Giancarlo. Table compression. In M.Y. Kao, editor, *Encyclopedia of Algorithms*, pages 939–942. Springer, 2008.

[5] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[6] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

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

[8] P. Ferragina, R. Giancarlo, and G. Manzini. The engineering of a compression boosting library: Theory vs practice in BWT compression. In *Proc. 14th European Symposium on Algorithms (ESA '06)*, pages 756–767. Springer Verlag LNCS n. 4168, 2006.

[9] P. Ferragina, R. Giancarlo, and G. Manzini. The myriad virtues of wavelet trees. *Information and Computation*, 207:849–866, 2009.

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

[11] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc. 46th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 184–193, 2005.

[12] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and searching xml data via two zips. In *Proc. 15th International World Wide Web Conference (WWW)*, pages 751–760, 2006.

[13] P. Ferragina and R. Venturini. The compressed permuterm index. *ACM Transactions on Algorithms (to appear)*, 2009.

[14] R. Giancarlo, A. Restivo, and M. Sciortino. From first principles to the burrows and wheeler transform and beyond, via combinatorial optimization. *Theoretical Computer Science*, 387(3):236–248, 2007.

[15] R. Giancarlo and M. Sciortino. Optimal partitions of strings: A new class of Burrows-Wheeler compression algorithms. In *Proc. 14th Symposium on Combinatorial Pattern Matching (CPM '03)*, pages 129–143. Springer-Verlag LNCS n. 2676, 2003.

[16] P. G. Howard and J. S. Vitter. Analysis of arithmetic coding for data compression. *Information Processing Management*, 28(6):749–764, 1992.

[17] H. Kaplan, S. Landau, and E. Verbin. A simpler analysis of burrows-wheeler-based compression. *Theoretical Computer Science*, 387(3):220–235, 2007.

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

[19] P. Kulkarni, F. Douglis, J.D. LaVoie, and J.M. Tracey. Redundancy elimination within large collections of files. In *USENIX Annual Technical Conference*, pages 59–72, 2004.

[20] V. Mäkinen and G. Navarro. Position-restricted substring searching. In *Proc. 7th Latin American Symposium on Theoretical Informatics (LATIN)*, pages 703–714. Springer Verlag LNCS n. 3887, 2006.

[21] V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *Procs 14th Symp. on String Processing and Information Retrieval (SPIRE)*, pages 229–241. Springer Verlag LNCS n. 4726, 2007.

[22] G. Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001.

[23] A. Moffat and R.Y. Isal. Word-based text compression using the burrows-wheeler transform. *Information Processing Management*, 41(5):1175–1192, 2005.

[24] Z. Ouyang, N.D. Memon, T. Suel, and D. Trendafilov. Cluster-based delta compression of a collection of files. In *Procs 3rd Conference on Web Information Systems Engineering (WISE)*, pages 257–268. IEEE Computer Society, 2002.

[25] T. Suel and N. Memon. Algorithms for delta compression and remote file synchronization. In Khalid Sayood, editor, *Lossless Compression Handbook*. Academic Press, 2002.

[26] D. Trendafilov, N. Memon, and T. Suel. Compressing file collections with a TSP-based approach. Technical report, Technical Report TR-CIS-2004-02, Polytechnic University, 2004.

[27] B.D. Vo and K.-P. Vo. Compressing table data with column dependency. *Theoretical Computer Science*, 387(3):273–283, 2007.

[28] 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.

Appendix A
An example for the booster

In this section we prove that there exists an infinite class of strings for which the partition selected by booster is far from the optimal one by a factor $\Omega(\sqrt{\log n})$. Consider an alphabet $\Sigma = \{c_1, c_2, \dots, c_\sigma\}$ and assume that $c_1 < c_2 < \dots < c_\sigma$. We divide it into $l = \sigma/\alpha$ groups of α consecutive symbols each, where $\alpha > 0$ will be defined later. Let $\Sigma_1, \Sigma_2, \dots, \Sigma_l$ denote these sub-alphabets. For each Σ_i, we build a De Bruijn sequence T_i in which each pair of symbols of Σ_i occurs exactly once. By construction each sequence T_i has length α^2. Then, we define $T = T_1 T_2 \dots T_l$, so that $|T| = \sigma\alpha$ and each symbol of Σ occurs exactly α times in T. Therefore, the first column of BWT matrix is equal to $(c_1)^\alpha (c_2)^\alpha \dots (c_\sigma)^\alpha$. We denote with L_c the portion of $\mathtt{BWT}(T)$ that has symbol c as prefix in the BWT matrix. By construction, if $c \in \Sigma_i$, we have that any L_c has either one occurrence of each symbol of Σ_i or one occurrence of these symbols of Σ_i minus one plus one occurrence of some symbol of Σ_{i-1} (or Σ_l if $i = 1$). In both cases, each L_c has α symbols, which are all distinct. Notice that by construction, the longest common prefix among any two suffixes of T is at most 1. Therefore, since the booster can partition only using prefix-close contexts (see [10]), there are just three possible partitions: (1) one substring containing all symbols of L, (2) one substring per L_c, or (3) as many substrings as symbols of L. Assuming that the cost of each model is at least $\log \sigma$ bits[5], then the costs of all possible booster's partitions are:

1. Compressing the whole L at once has cost at least $\sigma\alpha \log \sigma$ bits. In fact, all the symbols in Σ have the same frequency in L.
2. Compressing each string L_c costs at least $\alpha \log \alpha + \log \sigma$ bits, since each L_c contains α distinct symbols. Thus, the overall cost for this partition is at least $\sigma\alpha \log \alpha + \sigma \log \sigma$ bits.
3. Compressing each symbol separately has overall cost at least $\sigma\alpha \log \sigma$ bits.

We consider the alternative partition which is not achievable by the booster that subdivides L into σ/α^2 substrings denoted $S_1, S_2, \dots, S_{\sigma/\alpha^2}$ of size α^3 symbols each (recall that $|T| = \sigma\alpha$). Notice that each S_i is drawn from an alphabet of size smaller than α^3.

The strings S_i are compressed separately. The cost of compressing each string S_i is $O(\alpha^3 \log \alpha^3 + \log \sigma) = O(\alpha^3 \log \alpha + \log \sigma)$. Since there are σ/α^2 strings S_is, the cost of this partition is $P = O(\sigma\alpha \log \alpha + (\sigma/\alpha^2) \log \sigma)$. Therefore, by setting $\alpha = O(\sqrt{\log \sigma} / \log \log \sigma)$, we have that $P = O(\sigma \sqrt{\log \sigma})$ bits. As far as the booster is concerned, the best compression is achieved by its second partition whose cost is $O(\sigma \log \sigma)$ bits. Therefore, the latter is $\Omega(\sqrt{\log \sigma})$ times larger than our proposed partition. Since $\sigma \geq \sqrt{n}$, the ratio among the two partitions is $\Omega(\sqrt{\log n})$.

[5]Here we assume that it contains at least one symbol. Nevertheless, as we will see, the compression gap between booster's partition and the optimal one grows as the cost of the model becomes bigger.

Appendix B
Proof of Theorem 5

We describe a data structure supporting operations $\mathtt{Append}(w)$ and $\mathtt{Remove}()$ when the base compressor is BWT-based, and the input text T is the concatenation of a sequence of pages $S[1], S[2], \ldots, S[m]$ separated by unique separator symbols $\#_1, \#_2, \ldots, \#_m$, which are not part of Σ and are lexicographically larger than any symbol in Σ. We assume that the separator symbols in the $\mathtt{BWT}(T)$ are ignored by the MTF step, which means that when the MTF encoder finds a separator in $\mathtt{BWT}(T)$, this is replaced with the corresponding integer without altering the MTF-list. This variant does not introduce any compression penalty (because every separator occurs just once) but simplifies the discussion that follows. We denote with $sa_T[1,n]$ and $isa_T[1,n]$ respectively the suffix array of T and its inverse. Given a range $I = [a,b]$ of positions of T, an occurrence of a symbol of $\mathtt{BWT}(T)$ is called $active_{[a,b]}$ if it corresponds to a symbol in $T[a,b]$. For any range $[a,b] \subset [n]$ of positions in T, we define $\mathtt{RBWT}(T[a,b])$ as the string obtained by concatenating the $\text{active}_{[a,b]}$ symbols of $\mathtt{BWT}(T)$ by preserving their relative order. In the following, we will not indicate the interval when it will be clear from the context. Notice that, due to the presence of separators, $\mathtt{RBWT}(T[a,b])$ coincides with $\mathtt{BWT}(T[a,b])$ when $T[a,b]$ spans a group of contiguous pages (see [13] and references therein). Moreover, $\mathtt{MTF}(\mathtt{RBWT}(T[a,b]))$ is the string obtained by performing the MTF algorithm on $\mathtt{RBWT}(T[a,b])$. We will call the symbol $\mathtt{MTF}(\mathtt{RBWT}(T[a,b]))[i]$ as the MTF-encoding of the symbol $\mathtt{RBWT}(T[a,b])[i]$.

For each window w, our solution will not explicitly store neither $\mathtt{RBWT}(w)$ or $\mathtt{MTF}(\mathtt{RBWT}(T[a,b]))$ since this might require a superlinear amount of space. Instead, we maintain only an array F_w of size σ whose entry $F_w[e]$ keeps the number of occurrences of the encoding e in $\mathtt{MTF}(\mathtt{RBWT}(w))$. The array F_w is enough to compute the 0-order entropy of $\mathtt{MTF}(\mathtt{RBWT}(w))$ in σ time (or eventually the exact cost of compressing it with Huffman in $\sigma \log \sigma$ time).

We are left with showing how to correctly keep updated F_w after a $\mathtt{Remove}()$ or an $\mathtt{Append}(w)$. In the following we will concentrate only on $\mathtt{Append}(w)$ since $\mathtt{Remove}()$ is symmetrical. The idea underlying the implementation of $\mathtt{Append}(w)$, where $w = S[l,r]$, is to *conceptually insert* the symbols of the next page $S[r+1]$ into $\mathtt{RBWT}(w)$ one at time from left to right. Since the relative order among the symbols of $\mathtt{RBWT}(w)$ is preserved in $\mathtt{BWT}(T)$, it is more convenient to work with active symbols of $\mathtt{BWT}(T)$ by resorting to a data structure, whose details are given later, which is able to efficiently answer the following two queries with parameters c, I and h, where $c \in \Sigma$, $I = [a,b]$ is a range of positions in T and h is a position in $\mathtt{BWT}(T)$:

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

This data structure is built over the whole text T and requires $O(|T|)$ space.

Let c be the symbol of $S[r_i+1]$ we have to conceptually insert in $\mathtt{RBWT}(T[a,b])$. We can compute the position (say, h) of this symbol in $\mathtt{BWT}(T)$ by resorting to the inverse suffix array of T. Once we know position h, we have to determine what changes in $\texttt{MTF}(\texttt{RBWT}(w))$ the insertion of c has produced and update F_w accordingly. It is not hard to convince ourselves that the insertion of symbol c changes no more than σ encodings in $\texttt{MTF}(\texttt{RBWT}(w))$. In fact, only the first active occurrence of each symbol in Σ after position h may change its $\texttt{MTF}$ encoding. More precisely, let h_p and h_n be respectively the last active occurrence of c before h and the first active occurrence of c after h in $\texttt{BWT}(w)$, then the first active occurrence of a symbol after h changes its $\texttt{MTF}$ encoding if and only if it occurs active both in $\texttt{BWT}(w)[h_p, h]$ and in $\texttt{BWT}(w)[h, h_n]$. Otherwise, the new occurrence of c has no effect on its $\texttt{MTF}$ encoding. Notice that h_p and h_n can be computed via proper queries $\texttt{Prev}_c$ and $\texttt{Next}_c$. 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. Once we have these positions, we can recover the status of the $\texttt{MTF}$ list, denoted λ, before encoding c at position h. This is simply obtained by sorting the symbols ordered by decreasing position. In the second step, for each distinct symbol that occurs active in $\texttt{BWT}(w)[h_p, h]$, we find its first active occurrence in $\texttt{BWT}(w)[h, h_n]$. Knowing λ and these occurrences sorted by increasing position, we can simulate the $\texttt{MTF}$ algorithm to find the old and new encodings of each of those symbols.

This provides an algorithm to perform $\texttt{Append}(w)$ by making $O(\sigma)$ queries of types $\texttt{Prev}$ and $\texttt{Next}$ for each symbol of the page to append in w. To complete the proof of the time bounds in Theorem 5 we have to show how to support queries of type $\texttt{Prev}$ and $\texttt{Next}$ in $O(\log n)$ time and $O(n)$ space. 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 the set $[n] \times [n]$ (notice that n can be larger than p), such that no pair of points shares the same x- or y-coordinate, there exists a data structure [20] requiring $O(p)$ space and supporting the following two queries in $O(\log p)$ time:

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

Initially we compute isa_T and sa_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 $\texttt{rangemax}(I, isa_T[h+1])$ on the set P_c, and the same holds for $\texttt{Next}_c$ by using $\texttt{rangemin}$ instead of $\texttt{rangemax}$, this completes the reduction and the proof of the theorem.