On Optimally Partitioning a Text to Improve Its Compression*

Paolo Ferragina, Igor Nitto, and Rossano Venturini

Department of Computer Science, University of Pisa
{ferragina,nitto,rossano}@di.unipi.it

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 further elaborated and extended to strings and trees by [10,11,20], but it is still open how to efficiently compute the optimal partition [4]. In this paper we provide the first algorithm which is guaranteed to compute in $O(n\,\texttt{polylog}(n))$ time a partition of T whose compressed output is guaranteed to be no more than $(1+\epsilon)$-worse the optimal one, where ϵ is any positive constant.

1 Introduction

Reorganizing data in order to improve the performance of a given compressor $\mathcal{C}$ is an important paradigm of 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 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 which 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 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 [6,23], etc.), different permutations (see e.g. [14,25]), different base compressors to be boosted (0-th order compressors, `gzip`, `bzip2`, etc.). Among these plethora

* It has been partially supported by Yahoo! Research, and two Italian MIUR FIRB Projects: Italy-Israel and 2006-Linguistica. The authors' address is Dipartimento di Informatica, L.go B. Pontecorvo 3, 56127 Pisa, Italy.

A. Fiat and P. Sanders (Eds.): ESA 2009, LNCS 5757, pp. 420–431, 2009.

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 them in such a way that individually compressing contiguous groups of columns 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 for the A-TSP 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 fast and very simply heuristics.

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. [26]). 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. Several papers have given an analytic account of this phenomenon [21,9,17,20] and have shown, also experimentally [8], that the partitioning of the BW-transformed text is a key step for achieving effective compression ratios. Starting from these premises, [15] attacked the computation of the optimal partition of T via a DP-approach, which turned out to be very costly; then [10] (and subsequently many other authors, see e.g. [9,20,11]) proposed a *boosting* solution which is *not* optimal but, nonetheless, achieves interesting k-th order-entropy bounds. This is indeed a subtle point, frequently neglected. In fact, we are able to show [12] that there exists an infinite class of strings for which the compression achieved by the *booster* is far from the optimal-partitioning by a multiplicative factor $\Omega(\sqrt{\log n})$.

There is another interesting scenario in which the PPC-paradigm occurs and this is when T is a single (long) file, eventually obtained by concatenating a collection of (smaller) 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 e.g. [23,6]. In these cases, the partitioning step looks for *homogeneous* groups of contiguous files (or even within-file blocks of chars) which can be effectively compressed together by a base-compressor $\mathcal{C}$ — typically `gzip`, `bzip2`, `ppm`, etc. [26]. It is clear that how much redundancy can be detected and exploited by these compressors depends on their ability to "look back" at the previously seen data, and this has a cost in terms of memory usage and running time. Thus most compression systems provide a facility that controls the amount of data that may be processed at once — usually called the *block size* — ranging from few hundreds KBs [26] to few hundreds MBs [8]. Subtly, 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). A similar example can be shown [12] for more powerful compressors, such as the k-th order entropy encoder `ppm` which compresses each symbol according to its preceding k-long context. Therefore the impact of the choice of the block size cannot be underestimated and may be problematic, since it may change along the whole file we are compressing.

In summary, fast and effective algorithms or heuristics for the permuting step are known (see e.g. [3,10,11,24]), but not yet known is how to compute *efficiently* the *optimal* partition of the permuted data for compression boosting (see [4]). The goal of this paper is to provide 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 $T = T_1T_2 \cdots T_k$, 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 $\mathcal{P}_{\texttt{opt}}$ which achieves the shortest compressed output, namely $\mathcal{P}_{\texttt{opt}} = \min_{\mathcal{P}} \texttt{Cost}(\mathcal{P})$. As we mentioned above, $\mathcal{P}_{\texttt{opt}}$ might be computed via a Dynamic-Programming approach [3,15] in $\Theta(n^3)$ time, which is clearly unfeasible even on small input sizes n. In this paper we provide the first algorithm which is guaranteed to compute in $O(n(\log_{1+\epsilon} n)\ \texttt{polylog}(n)))$ time a partition of T whose compressed output is guaranteed to be no more than $(1 + \epsilon)$-worse than the optimal one, where ϵ may be any positive constant. Due to the lack of space, proofs and many details are omitted from this paper but can be found in [12].

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 $H_0(T) = \sum_{c \in \Sigma}^{h} \frac{n_c}{n} \log \frac{n}{n_c}$. 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 [26]. The so-called zero-th order statistical compressors (such as Huffman or Arithmetic [26]) 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 (*semi-static* 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 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 $\mathcal{C}$. We will also assume that $f_0(n,\sigma)$ can be computed in constant time. As an example, Huffman has $f_0(n,\sigma) = \sigma \log \sigma + n$ and $\lambda = 1$, whereas Arithmetic has $f_0(n,\sigma) = \sigma \log n + \log n/n$ and $\lambda = 1$.

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 `si` in T are followed by the symbols `s` and `p`, respectively. The k-th order *empirical* entropy of T is defined as $H_k(T) = \frac{1}{|T|} \sum_{u \in \Sigma^k} |u_T|\, H_0(u_T)$. We have $H_k(T) \geq H_{k+1}(T)$ for any $k \geq 0$. As usual in data compression [21], 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. [18,21,10,9,20,11] and refs therein) authors have provided *upper bounds* in terms of $H_k(T)$ for sophisticated data-compression algorithms, such as `gzip` [18], `bzip2` [21,10,17], 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 [20] has $\lambda = 1$ and $f(|T|,\sigma) = O(\sigma^{k+1} \log |T| + |T| \log \sigma \log\log |T| / \log |T|)$.[1]

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 (Sect 3) and introduce few novel technicalities (Sect 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 Sect 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:

[1] Many results (see [21,10,9] and refs therein) are expressed in term of k-th order modified empirical entropy of T and have the form $\lambda |T| H_k^*(T) + g_k(\sigma)$. In [12] we show how to extend our results to this entropy too.

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 equals 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 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 (Sect 3.1). Second, we show some sufficient properties that $\mathcal{C}$ needs to satisfy in order to compute every edge's cost efficiently. These properties hold for some well-known compressors— e.g. 0-order compressors, `PPM`-like and `bzip`-like compressors— so, for them, we will show how to compute each edge cost in constant or polylogarithmic time (Sect 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 property 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 $O(n \log_{1+\epsilon} n)$. Hereafter, we 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 $d_{\mathcal{G}(T)}(v_j, v_q) \leq d_{\mathcal{G}(T)}(v_i, v_q)$ and $d_{\mathcal{G}(T)}(v_i, v_j) \leq d_{\mathcal{G}(T)}(v_i, v_q)$.*

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

Theorem 2. *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 2 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 Sect 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. `Remove`() moves the starting position l of all windows one position to the right (i.e. $l+1$);
2. `Append`(w_i) moves the ending 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 $|\mathcal{C}(T[l, r_i])|$.

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 `Remove`() once to increment index l and, for each $t = 1, \ldots, \log_{1+\epsilon}(n)$, we call operation `Append`(w_t) until we find the largest r_t such that `Size`$(w_t) = c(v_l, v_{1+r_t}) \leq (1+\epsilon)^t$. 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 $c(v_l, v_{1+r_t})$ efficiently. Just one character is entering w_t to its right, so we need to deploy this fact for making the computation of `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`$(w_t) = |\mathcal{C}(w_t)|$. 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 `Append` + `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 `Remove`, and $O(n \log_{1+\epsilon} n)$ calls to operations `Append` and `Size`.

Theorem 3. *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 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 Sect 2). The key point is actually to show how to efficiently compute `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 Sect 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 Sect 2) as we will do in Sect 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. We can extend the previous scheme to obtain an efficient solution up to $\sigma = O(\texttt{poly}(n))$ but, because of space limitations, its technicalities are reported in the full version of this paper [12]. The following lemma states the final result.

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 Sect 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 Sect 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]. An approach similar to the previous one (but with little more technicalities, given in [12]) can be used to solve the case in which we use a 0-th order adaptive compressor $\mathcal{C}$ for providing the edge-costs of $\mathcal{G}(T)$. For this case we can prove the same time and space bounds of Lemma 2. Combining this with Theorem 3 we obtain:

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 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 this result can be applied to the compression booster of [10] to fast obtain an approximation of the optimal partition of $\texttt{BWT}(T)$. This may be better than the algorithm of [10] both in time complexity, since that algorithm takes $O(n\sigma)$ time, and in compression ratio (details in [12]). The case of a large alphabet (namely, $\sigma = \Omega(\texttt{polylog}(n))$) is particularly interesting whenever we consider either a word-based BWT [22] or the XBW-transform over labeled trees [10] for which Σ is either the set of words or tags in a text. We notice that our result is interesting also for the *Huffword* compressor which is the standard choice for the storage of Web pages [26]; 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. Here $\texttt{Size}(w_i)$ must compute $|\mathcal{C}(T[l,r_i])|$ which is estimated, as detailed in Sect 2, 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 $T_q[i]$ is equal to the q-gram $T[i, i+q-1]$. Actually, we can remap the symbols of T_q to integers in $[1,n]$. In fact the number of distinct q-grams occurring in T_q is less than $n = |T|$. 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.

It is well-known that the k-th order entropy of a string (see definition Sect 2) can be expressed as the difference between the zero-th order entropy of its $(k+1)$-grams and the one of 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 4 (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* $\texttt{Size}()$ *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 $R(n) = O(\log_{1+\epsilon} n)$ 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 [12]).

Theorem 5. *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 Sect 2 we know entropy-bounded estimates for the output size of BWT-based compressors. So we could apply Theorem 5 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 $\texttt{BWT}(T)$-string. Typically, BWT is encoded via a sequence of simple compressors such as MTF, RLE (which is optional), and finally a 0-order encoder like Huffman or Arithmetic [26]. For each of these compression steps, a 0-th entropy bound is known [10,9], but the combination of these bounds may result far from the final compressed size produced by the overall sequence of compressors in practice [8].

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 5, but with the advantage of computing the $(1+\epsilon)$-optimal solution wrt the real compressed size, thus without any estimation of it by entropy-cost functions. Since in practice it is $\sigma = \texttt{polylog}(n)$, this slowdown is negligible. In order to achieve this result, we need to address a slightly different 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, in that 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 compressed output-size is a factor at most $(1+\epsilon)$ the minimum possible one, 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 an analog of Theorem 3 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 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 $w' = \texttt{MTF}(\texttt{BWT}(w))$ 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 [26]. 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 $\texttt{MTF}(s')$ differs from $\texttt{MTF}(s)$ 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 to [12] for details, and state here the result we are able to achieve.

Theorem 6. *Given a sequence of texts of total length n and alphabet size $\sigma = \texttt{poly}(n)$, 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.*

We conclude this paper by devising two possible directions of research, which consist either in investigating the design of $o(n^2)$-time algorithms for computing the *exact* optimal partition, and/or experimenting our solution over large textual datasets.

References

1. Bentley, J.L., McIlroy, M.D.: Data compression with long repeated strings. Information Sciences 135(1-2), 1–11 (2001)
2. 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. ACM-SIAM SODA, pp. 175–184 (2000)
3. Buchsbaum, A.L., Fowler, G.S., Giancarlo, R.: Improving table compression with combinatorial optimization. J. ACM 50(6), 825–851 (2003)

4. Buchsbaum, A.L., Giancarlo, R.: Table compression. In: Kao, M.Y. (ed.) Encyclopedia of Algorithms, pp. 939–942. Springer, Heidelberg (2008)
5. Burrows, M., Wheeler, D.: A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation (1994)
6. Chang, F., Dean, J., Ghemawat, S., et al.: Bigtable: A distributed storage system for structured data. ACM Trans. Comput. Syst. 26(2) (2008)
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. The MIT Press, McGraw-Hill Book Company (2001)
8. Ferragina, P., Giancarlo, R., Manzini, G.: The engineering of a compression boosting library: Theory vs practice in BWT compression. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 756–767. Springer, Heidelberg (2006)
9. Ferragina, P., Giancarlo, R., Manzini, G.: The myriad virtues of wavelet trees. Information and Computation 207, 849–866 (2009)
10. Ferragina, P., Giancarlo, R., Manzini, G., Sciortino, M.: Boosting textual compression in optimal linear time. J. ACM 52, 688–713 (2005)
11. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Structuring labeled trees for optimal succinctness, and beyond. In: Proc. FOCS, pp. 184–193 (2005)
12. Ferragina, P., Nitto, I., Venturini, R.: On optimally partitioning a text to improve its compression. CoRR, abs/0906.4692 (2009)
13. Ferragina, P., Venturini, R.: The compressed permuterm index. ACM Transactions on Algorithms (to appear, 2009)
14. Giancarlo, R., Restivo, A., Sciortino, M.: From first principles to the burrows and wheeler transform and beyond, via combinatorial optimization. Theoretical Computer Science 387(3), 236–248 (2007)
15. Giancarlo, R., Sciortino, M.: Optimal partitions of strings: A new class of Burrows-Wheeler compression algorithms. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 129–143. Springer, Heidelberg (2003)
16. Howard, P.G., Vitter, J.S.: Analysis of arithmetic coding for data compression. Information Processing Management 28(6), 749–764 (1992)
17. Kaplan, H., Landau, S., Verbin, E.: A simpler analysis of burrows-wheeler-based compression. Theoretical Computer Science 387(3), 220–235 (2007)
18. Kosaraju, R., Manzini, G.: Compression of low entropy strings with Lempel–Ziv algorithms. SIAM Journal on Computing 29(3), 893–911 (1999)
19. Kulkarni, P., Douglis, F., LaVoie, J.D., Tracey, J.M.: Redundancy elimination within large collections of files. In: USENIX, pp. 59–72 (2004)
20. Mäkinen, V., Navarro, G.: Implicit compression boosting with applications to self-indexing. In: Ziviani, N., Baeza-Yates, R. (eds.) SPIRE 2007. LNCS, vol. 4726, pp. 229–241. Springer, Heidelberg (2007)
21. Manzini, G.: An analysis of the Burrows-Wheeler transform. J. ACM 48(3), 407–430 (2001)
22. Moffat, A., Isal, R.Y.: Word-based text compression using the Burrows-Wheeler transform. Information Processing Management 41(5), 1175–1192 (2005)
23. Suel, T., Memon, N.: Algorithms for delta compression and remote file synchronization. In: Lossless Compression Handbook. Academic Press, London (2002)
24. Trendafilov, D., Memon, N., Suel, T.: Compressing file collections with a TSP-based approach. Technical report, TR-CIS-2004-02, Polytechnic University (2004)
25. Vo, B.D., Vo, K.-P.: Compressing table data with column dependency. Theoretical Computer Science 387(3), 273–283 (2007)
26. Witten, I.H., Moffat, A., Bell, T.C.: Managing Gigabytes: Compressing and Indexing Documents and Images. Morgan Kaufmann, San Francisco (1999)