Boosting Textual Compression in Optimal Linear Time

PAOLO FERRAGINA

Università di Pisa, Pisa, Italy

RAFFAELE GIANCARLO

Università di Palermo, Palermo, Italy

GIOVANNI MANZINI

Università del Piemonte Orientale, Alessandria, Italy

AND

MARINELLA SCIORTINO

Università di Palermo, Palermo, Italy

Dedicated to Antonio Restivo, on the occasion of his 60th birthday

Abstract. We provide a general boosting technique for Textual Data Compression. Qualitatively, it takes a good compression algorithm and turns it into an algorithm with a better compression

Extended abstracts related to this article appeared in *Proceedings of CPM 2001* and *Proceedings of ACM-SIAM SODA 2004*, and were combined due to their strong relatedness and complementarity.

The work of P. Ferragina was partially supported by the Italian MIUR projects "Algorithms for the Next Generation Internet and Web: Methodologies, Design and Experiments (ALGO-NEXT)", "Enhanced Content Delivery (ECD)", and "Grid.it".

The work of R. Giancarlo was partially supported by the Italian MIUR projects "Bioinformatica per la Genomica e la Proteomica", and "Metodi Combinatori ed Algoritmici per la Scoperta di Patterns in Biosequenze". Additional support provided by a CNRS Fellowship, France, sponsoring visits to the Institut Gaspard Monge, Marne La Vallee, France.

The work of G. Manzini was partially supported by the Italian MIUR projects "Algorithmics for Internet and the Web (ALINWEB)" and "Enhanced Content Delivery (ECD)".

The work of M. Sciortino was partially supported by the Italian MIUR projects "Linguaggi Formali ed Automi: Teoria ed Applicazioni", and "Linguaggi Formali ed Automi: Metodi, Modelli ed Applicazioni".

Authors' addresses: P. Ferragina, Dipartimento di Informatica, Largo B. Pontecorvo 3, I-56127 Pisa, Italy, e-mail: ferragina@di.unipi.it; R. Giancarlo and M. Sciortino, Dipartimento di Matematica ed Applicazioni, Via Archirafi 34, I-90123 Palermo, Italy, e-mail: {raffaele; mari}@math.unipa.it; G. Manzini, Dipartimento di Informatica, Via Bellini 25g, I-15100 Alessandria, Italy, e-mail: manzini@mfn.unipmn.it.

performance guarantee. It displays the following remarkable properties: (a) it can turn *any memoryless* compressor into a compression algorithm that uses the "best possible" contexts; (b) it is very simple and *optimal* in terms of time; and (c) it admits a decompression algorithm again optimal in time. To the best of our knowledge, this is the first boosting technique displaying these properties.

Technically, our boosting technique builds upon three main ingredients: the Burrows–Wheeler Transform, the Suffix Tree data structure, and a greedy algorithm to process them. Specifically, we show that there exists a proper partition of the Burrows–Wheeler Transform of a string s that shows a deep combinatorial relation with the kth order entropy of s. That partition can be identified via a greedy processing of the suffix tree of s with the aim of minimizing a proper objective function over its nodes. The final compressed string is then obtained by compressing individually each substring of the partition by means of the base compressor we wish to boost.

Our boosting technique is inherently combinatorial because it does not need to assume any prior probabilistic model about the source emitting s, and it does not deploy any training, parameter estimation and learning. Various corollaries are derived from this main achievement. Among the others, we show analytically that using our booster, we get better compression algorithms than some of the best existing ones, that is, LZ77, LZ78, PPMC and the ones derived from the Burrows–Wheeler Transform. Further, we settle analytically some long-standing open problems about the algorithmic structure and the performance of BWT-based compressors. Namely, we provide the first family of BWT algorithms that do not use Move-To-Front or Symbol Ranking as a part of the compression process.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management; E.4 [**Coding and Information Theory**]: *Data compaction and compression*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems; G.2.1 [**Discrete Mathematics**]: Combinatorics—*Combinatorial algorithms*; H.1.1 [**Models and Principles**]: Systems and Information theory; H.2.7 [**Database Management**]: Database Administration—*Data warehouse and repository*; H.3.2 [**Information Storage and Retrieval**]: Information Storage.

General Terms: Algorithms, Design, Theory

Additional Key Words and Phrases: Arithmetic coding, Burrows–Wheeler transform, empirical entropy, Huffman coding, Lempel–Ziv compressors, suffix tree, text compression

1. *Introduction*

A boosting technique, in very informal terms, can be seen as a method that, when applied to a particular class of algorithms, yields improved algorithms in terms of one or more parameters characterizing their performance in the class. For instance, Deterministic Amplification allows to take an RP or BPP algorithm, with some error probability, and to produce a new algorithm with, usually, an exponentially smaller error probability, at the expense of some increase in the number of random bits. The first such result was obtained by Karp et al. [1985] and has been followed by a series of very deep and significant findings in the area of Randomized Algorithms and Amplification of Weakly Random Sources (see, for instance, Nisan and Ta-Shma [1999] and Trevisan [2001]). Another remarkable example comes from Computational Learning Theory [Valiant 1984]. The first boosting technique in this field was obtained by Schapire [1990], who showed how to take a Weak Learning Algorithm and substantially increase its "prediction accuracy", again at some expenses in terms of time and space. The fundamental contribution by Schapire has lead, over the years, to the settlement of many fundamental problems in Computational Learning Theory and to a broad range of results, spanning many fields [Schapire 2002].

As the two examples just given indicate, general boosting techniques have a deep significance for Computer Science. Indeed, for one thing, they provide guidelines to the practitioners on how to obtain high-quality algorithms out of good algorithms, making their engineering task somewhat simpler. They also provide quite a few insights into the class of problems to which they can be applied. Unfortunately, they are also very hard to come by.

Data Compression is a key component of efficient Networks and Computer Systems. However, despite the massive literature produced in this field, ranging from the foundations laid by Shannon to the more practical engineering aspects [Storer 1992], we are not aware of any boosting technique for data compression algorithms. In this article, we provide a general boosting technique for Data Compression and in particular, for text compression, that is, strings of symbols over an alphabet (see Witten et al. [1999, Chap. 2]). Given the deep connection between Prediction and Compression [Rissanen 1984], the results from Computational Learning Theory may hint at the existence of boosting techniques for Textual Data Compression, given sufficient data characterizing the statistical properties of the Source. Unfortunately, to the best of our knowledge, we are not aware of any such algorithm. On the other hand, our technique is inherently combinatorial: it *does not need* to assume any prior probabilistic model about the source emitting the string and there is no training, parameter estimation and learning. Even more remarkably, it displays the following properties: (a) it can turn *any memoryless* compressor into a compression algorithm that uses the "best possible" contexts; (b) it is very simple and *optimal* in terms of time; (c) it admits a decompression algorithm again optimal in time; (d) it can be shown analytically that it yields better algorithms than some of the best existing ones, that is, LZ77 [Ziv and Lempel 1977], LZ78 [Ziv and Lempel 1978], PPMC [Moffat 1990] and the one proposed by Burrows and Wheeler, based on their Transform [Burrows and Wheeler 1994; Manzini 2001].

The significance of our main result is twofold. From the practical side, we open the way, starting from solid theoretic ground, to novel compressors that deserve a careful experimental study and which, in turn, can have a great practical impact. From the theoretical side, we analytically settle a long-standing open problem posed by Fenwick [1996] about the algorithmic structure of compressors based on the Burrows–Wheeler Transform (BWT from now on). We also show that the BWT is neither a necessary nor a sufficient component of our boosting technique, which indeed deploys a novel and deep combinatorial relation between Entropy, the BWT, and the Suffix Tree data structure [McCreight 1976]. This latter methodological finding should not be underestimated because, despite a few theoretical [Effros et al. 2002; Manzini 2001; Sadakane 1998] and many experimental studies [Arnavut 2002; Balkenhol et al. 1999; Deorowicz 2002; Fenwick 1996; Larsson 1998; Seward 1997; Wirth and Moffat 2001], insights into the nature and the power of the "magic box" BWT have been so far very evasive.

1.1. STATEMENT OF RESULTS. Let s be a string drawn from a constant size alphabet Σ. Given an integer k, let $H_k(s)$ denote the kth order *empirical entropy* of s. Moreover, let $H_k^*(s)$ be the kth order *modified empirical entropy* of s. Both functions are defined in Section 2.1. As pointed out in Manzini [2001], the modified empirical entropy H_k^* is a more realistic measure to bound the worst-case performance of compression algorithms, since its value is at least equal to the minimum number of bits needed to write down the length of the input string. Throughout this article, k denotes an integer and the alphabet Σ has constant size.

Recall that, given an input string s, the Burrows–Wheeler Transform (BWT) produces a permuted string bwt(s). For the time being, all we need to know is that the

BWT is invertible, that is, from $\mathtt{bwt}(s)$ we can recover s, and that its computation is basically a sorting of all suffixes of s.

Consider a compression algorithm A that squeezes any string $z\#$ in at most $\lambda|z|H_0(z) + \eta|z| + \mu$ bits, where λ, μ and η are constants independent of z and # is a special symbol not appearing elsewhere in z. Our boosting technique consists of three steps:

(1) compute $\hat{s} = \mathtt{bwt}(s^R)$, that is, the BWT of the string s reversed;
(2) using the suffix tree of s^R, greedily partition $\hat{s}$ so that a suitably defined objective function is minimized;
(3) compress each substring of the partition, separately, using algorithm A.

We will show that, for any $k \geq 0$, the length in bits of the string resulting from the boosting is bounded by:

$$\lambda|s|H_k(s) + \log_2|s| + \eta|s| + g_k. \tag{1}$$

Assume now that A squeezes any string $z\#$ in at most $\lambda|z|H_0^*(z) + \mu$ bits. We will show that, for any $k \geq 0$, the length in bits of the compressed string resulting from boosting is bounded by:

$$\lambda|s|H_k^*(s) + \log_2|s| + g_k. \tag{2}$$

In both bounds (1) and (2), g_k is a constant that depends only on k and not on the string s. We also remark that both bounds hold simultaneously for all k, that is, k is not a parameter known to the algorithm.

At this point, the effect of boosting should be clear. One can take a compressor achieving a good performance in terms of the 0th order entropy and, via our boosting technique, obtain a new compressor with a performance guarantee bounded in terms of the kth order entropy, simultaneously for all k. Putting it another way, one can take a compression algorithm that uses no context information at all and, via the boosting process, obtain an algorithm that automatically "discovers" and effectively uses the "best possible" contexts. Moreover, the boosting is optimal in time and introduces a space overhead of $O(|s|\log|s|)$ bits with respect to A.[1] So, the design of effective algorithms that use context information, or memory, reduces to the one of designing memoryless algorithms. This is the first main achievement of our article.

We also investigate whether the BWT is either a necessary or a sufficient tool for designing our booster. We show that neither of these two cases holds. In fact, if we use in Step (1) of our booster, the so-called "bounded context" transform, where only contexts of length up to some fixed k_0 are considered [Schindler 1997], then the same results hold but for all $k \leq k_0$. Actually, there might exist other transforms that are substantially different from the BWT and could find effective application within our boosting technique. Recently, it has been shown by Crochemore et al. [2005] that the BWT is a particular case of a bijection due to Gessel and Reutenauer [1993], which allows the enumeration of permutations by descents and cyclic type. In this article, we show that, as long as those transforms are invertible and mappable

[1] The logarithmic term in the space occupancy comes from an exact count of the space needed to store the suffix tree data structure [McCreight 1976] and some integer quantities. This term is usually not accounted for with the Uniform Memory Model.

to a suffix tree [McCreight 1976] in accordance with some proximity constraint, our boosting results still hold.

At this point, it is natural to ask ourselves if the bounds (1) and (2) are the best possible we can aim for a booster. We show that no compression algorithm, satisfying some mild assumptions on its inner working, can achieve a bound of the form $|s|H_k^*(s) + g_k$, for any $k \geq 0$ and g_k constant. This specifically means that both the multiplicative factor λ and the additive logarithmic term in (1) and (2) cannot be dropped simultaneously. This result follows by a lower bound on the coding of the integers and settles an open problem raised in Manzini [2001].

Finally, we prove the applicability of our boosting technique by proposing two candidates for the base compressor A: the algorithms HC and RHC. These algorithms are such that for any string z it is $|\mathtt{HC}(z\#)| \leq |z|H_0(z) + |z| + \Theta(1)$ and $|\mathtt{RHC}(z\#)| \leq 2.5|z|H_0^*(z) + \Theta(1)$. When these algorithms are used together with our boosting technique, we obtain two compressors satisfying the following bounds, respectively,

$$|s|H_k(s) + |s| + \log_2 |s| + g_k \tag{3}$$

and

$$2.5|s|H_k^*(s) + \log_2 |s| + g_k. \tag{4}$$

The above worst-case bounds are better than the worst-case bounds proven (so far) for any other compressor, including the ones based on the BWT. Indeed, in their seminal paper, Burrows and Wheeler proposed to compress the output of the BWT using Move-to-Front Encoding [Bentley et al. 1986] (shortly MTF), followed by an order zero compressor (Arithmetic or Huffman coding [Cover and Thomas 1990]). In Manzini [2001], it is shown that if we use Arithmetic coding the above algorithm produces an output bounded by

$$8|s|H_k(s) + \frac{2}{25}|s| + \log_2 |s| \tag{5}$$

for any $k \geq 0$. Several variants of this first BWT-based compressor have been proposed in the literature. A common feature of these variants is that they use some form of Run-Length Encoding [Cover and Thomas 1990] (shortly RLE) at some point of the compression process. In Manzini [2001], it is shown that if we process the output of the BWT using MTF, followed by RLE, followed by Arithmetic coding, we get an output bounded by[2]

$$(5 + \epsilon)|s|H_k^*(s) + \log_2 |s| + g'_k \tag{6}$$

for any $k \geq 0$ and $\epsilon \approx 10^{-2}$. We note that the bound (3) is better than (5) in the leading entropy term, while the bound (4) is a factor of 2 improvement over (6). As far as other compressors are concerned, in Kosaraju and Manzini [1999] and Manzini [2001], it was shown that the bound (6) (and therefore (4)) cannot hold for many of them: LZ77 [Ziv and Lempel 1977], LZ78 [Ziv and Lempel 1978] and PPMC [Moffat 1990]. It remains an open problem whether a similar bound

[2] An attentive reader may have noticed the term $\log |s|$ in (5) and (6), which was not present in Manzini [2001]. The point is that the output of the Burrows–Wheeler Transform consists of a permutation of s and of an integer in the range $[1, |s|]$ (see Section 3). In Manzini [2001] the compression bounds refer only to the compression of the permutation of s. In this article, we account also for the space needed to encode the above integer.

can hold for DMC [Cormak and Horspool 1987] and PPM* [Cleary and Teahan 1997], for which no theoretical analysis (at least in terms of empirical entropy) is available.

Summing up, our boosting technique yields compression algorithms that are analytically superior to many of the best existing ones. We point out that, although based on BWT, our algorithm does not use MTF—or any other type of symbol ranking—as part of the compression process. This settles a long-standing open problem first raised by Fenwick [1996] about the existence of alternatives to MTF encoding in BWT-based compressors. Indeed, since the appearance of the first BWT-based compressors, the general feeling among theoreticians and practitioners was that the MTF coder introduces some degree of inefficiency [Arnavut 2002; Balkenhol et al. 1999; Deorowicz 2002; Wirth and Moffat 2001]. Our results prove analytically that an alternative to MTF does exist, and that such an alternative takes linear time and provides better compression bounds than MTF.

The article is organized as follows. Section 2 introduces some basic terminology on entropy and prefix codes as well as some useful notation. In Section 3, we recall the definition and the properties of the Burrows–Wheeler Transform (shortly BWT), and we describe the basic structure of a compression booster based on the BWT, highlighting the difficulties incurred in its design. In Section 4, we introduce the key idea underlying our boosting technique, namely a deep combinatorial relation among the entropy, the BWT and the Suffix Tree data structure. This relation materializes itself in the novel notion of *leaf cover* of a suffix tree. In Section 5, we detail our boosting technique and design a greedy algorithm for finding an optimal leaf cover whose cost is strictly related to the kth order entropy of the input string. Various corollaries on the nature and limitations of our boosting technique are also proven in this section. In Section 6, we show the applicability of our boosting technique by describing two 0th order compressors (HC and RHC) whose performance can be improved by our booster achieving the bounds (3) and (4). Finally, in Section 7, we investigate the improvability of our boosting bounds by settling, as a corollary, a conjecture posed in Manzini [2001]. We conclude our article with some comments on future directions of research.

2. *Empirical Entropies and Code Lengths*

In this section, we recall the definition of the empirical entropy of a string and of some of its variants. We also introduce information measures that can be seen as "dual" of empirical entropies. In agreement with classic Information Theory, all of the variants of empirical entropy we define are lower bounds on the output size of the encoding of a string, where the codeword of each symbol depends on k symbols "preceding it" in the string. Additional details on those information measures can be found in Cover and Thomas [1990] and Manzini [2001].

2.1. THE EMPIRICAL ENTROPY OF A STRING. We need to point out one important difference between empirical entropy and the entropy defined in the probabilistic setting. Shannon's entropy is an expected value taken on an ensemble of strings, while empirical entropy is defined *pointwise* for *any* string and can be used to measure the performance of compression algorithms as a function of the *string structure*, thus *without* any assumption on the input source. In a sense, compression bounds produced in terms of empirical entropy are worst-case measures.

Let s be a string over the alphabet $\Sigma = \{a_1, \ldots, a_h\}$ and, for each $a_i \in \Sigma$, let n_i be the number of occurrences of a_i in s. Throughout this paper we assume that $n_i \geq n_{i+1}$. Moreover, all logarithms are taken to the base 2 and we assume $0 \log 0 = 0$. Let $\{P_i = n_i/|s|\}_{i=1}^{h}$ be the empirical probability distribution for the string s. The *zeroth order empirical entropy* of the string s is defined as

$$H_0(s) = -\sum_{i=1}^{h} P_i \log(P_i). \tag{7}$$

For any length-k string w, we denote by $\vec{w_s}$ the string of single symbols following the occurrences of w in s, taken from left to right.

Example 2.1. Let $s = \texttt{mississippi}$, $w = \texttt{si}$. The two occurrences of si in s are followed by the symbols s and p, respectively. Hence, $\vec{w_s} = \texttt{sp}$.

The kth order empirical entropy of s is defined as:

$$H_k(s) = \frac{1}{|s|} \sum_{w \in \Sigma^k} |\vec{w_s}|\, H_0(\vec{w_s}). \tag{8}$$

Manzini [2001] argued that, for highly compressible strings, $|s|H_k(s)$ fails to provide a reasonable bound to the performance of compression algorithms. For that reason, he introduced the *zeroth order modified empirical* entropy:

$$H_0^*(s) = \begin{cases} 0 & \text{if}|s| = 0 \\ (1 + \lfloor \log |s| \rfloor)/|s| & \text{if}|s| \neq 0 \text{ and } H_0(s) = 0 \\ H_0(s) & \text{otherwise.} \end{cases} \tag{9}$$

Note that for a nonempty string s, $|s|H_0^*(s)$ is at least equal to the number of bits needed to write down the length of s in binary. The *kth order modified empirical entropy* H_k^* is then defined in terms of H_0^* as the maximum compression we can achieve by looking at *no more than k* symbols preceding the one to be compressed.[3] Formally, let $\mathcal{S}_k$ be a set of substrings of s having length *at most k*. We say that the set $\mathcal{S}_k$ is a *suffix cover* of Σ^k, and write $\mathcal{S}_k \preceq \Sigma^k$, if any string in Σ^k has a *unique suffix* in $\mathcal{S}_k$.

Example 2.2. Let $\Sigma = \{a, b\}$ and $k = 3$. Two suffix covers for Σ^3 are $\{a, b\}$ and $\{a, ab, abb, bbb\}$. Indeed, any string in Σ^3 has a unique suffix in both sets.

For any suffix cover $\mathcal{S}_k$, let

$$H_{\mathcal{S}_k}^*(s) = \frac{1}{|s|} \sum_{w \in \mathcal{S}_k} |\vec{w_s}|\, H_0^*(\vec{w_s}). \tag{10}$$

The value $H_{\mathcal{S}_k}^*(s)$ represents the compression we can achieve using the strings in $\mathcal{S}_k$ as contexts for the prediction of the next symbol. The entropy $H_k^*(s)$ is defined

[3] Note that H_k is defined in terms of H_0 as the maximum compression we can achieve by looking at *exactly k* symbols preceding the one to be compressed. For H_k^*, we instead consider H_0^* and a context of *at most k* symbols. This is to ensure that $H_{k+1}^*(s) \leq H_k^*(s)$ for any string s. See Manzini [2001] for details.

as the compression that we can achieve using a best possible suffix cover:

$$H_k^*(s) = \min_{\mathcal{S}_k \preceq \Sigma^k} H^*_{\mathcal{S}_k}(s). \tag{11}$$

In the following, we use $\mathcal{S}_k^*$ to denote a suffix cover for which the minimum of (11) is achieved. Therefore, we write

$$H_k^*(s) = \frac{1}{|s|} \sum_{w \in \mathcal{S}_k^*} |\overrightarrow{w_s}|\, H_0^*(\overrightarrow{w_s}). \tag{12}$$

Notice that, when $H_0(s) = 0$, $|s| H_0^*(s)$ gives the number of bits needed to store the length of s, which we need to encode. Unfortunately, in that definition there is the implicit assumption that $|s|$ is known both to the compression and decompression algorithms. That is usually not the case, since the decompression algorithm knows $|s|$ only if the compression algorithm encodes it, either explicitly or implicitly. This point will be further elaborated in Section 7, where we show that even the modified empirical entropy of any order may be a too demanding bound for compression algorithms to achieve.

In the rest of this article, we use the Burrows–Wheeler Transform as a key component of our compression booster. As we will see, the BWT relates substrings of s with the single symbols *preceding* their occurrences in s. Conversely, $H_k^*(s)$ relates substrings of s with the single symbols *following* their occurrences in s. In order to simplify the analysis of our algorithms, we introduce an additional notation offering this other point of view.

Let $\overleftarrow{w_s}$ be the string of single symbols that *precede* all occurrences of the substring w in the string s (cfr. the definition of $\overrightarrow{w_s}$). Let $\mathcal{P}_k$ be a set of strings, having length *at most k*, that are *unique prefixes* of all strings in Σ^k. We call $\mathcal{P}_k$ a *prefix cover* of Σ^k (cfr. the definition of suffix cover $\mathcal{S}_k$).

Example 2.3. Let $\Sigma = \{a, b\}$ and $k = 3$. Two prefix covers for Σ^3 are $\{a, b\}$ and $\{a, ba, bb\}$. Indeed, any string in Σ^3 has a unique prefix in both sets.

Substituting in (10) $\overrightarrow{w_s}$ with $\overleftarrow{w_s}$, and $\mathcal{S}_k$ with $\mathcal{P}_k$, we define for any prefix cover $\mathcal{P}_k$

$$\overleftarrow{H}^*_{\mathcal{P}_k}(s) = \frac{1}{|s|} \sum_{w \in \mathcal{P}_k} |\overleftarrow{w_s}|\, H_0^*(\overleftarrow{w_s}). \tag{13}$$

In analogy with (11), we take the minimum of (13) over all prefix covers. Let $\mathcal{P}_k^*$ denote a prefix cover which minimizes (13). Using $\mathcal{P}_k^*$, we define the function $\overleftarrow{H}_k^*$ which is analogous to the kth order empirical entropy (12), but now referring to preceding symbols

$$\overleftarrow{H}_k^*(s) = \frac{1}{|s|} \sum_{w \in \mathcal{P}_k^*} |\overleftarrow{w_s}|\, H_0^*(\overleftarrow{w_s}). \tag{14}$$

In analogy with $\overleftarrow{H}_k^*$, we define $\overleftarrow{H}_k$ substituting in (8) $\overrightarrow{w_s}$ with $\overleftarrow{w_s}$. The next lemma gives the relationship among all of those information measures. Let s^R denote the reversal of the string s.

LEMMA 2.4. *For any string s, and any integer $k \geq 0$, $\overleftarrow{H}_k(s) = H_k(s^R)$ and $\overleftarrow{H}^*_k(s) = H^*_k(s^R)$.*

PROOF. We give a proof only for $\overleftarrow{H}^*_k$, since the one for $\overleftarrow{H}_k$ follows along the same lines. The set of single symbols following w in s coincides with the set of single symbols preceding w^R in s^R. More precisely, $\overrightarrow{w_s}$ is the reverse of the string containing the symbols preceding w^R in s^R. Furthermore, if $\mathcal{S}_k$ is a suffix cover of Σ^k, then the reversal of the strings in $\mathcal{S}_k$ is a prefix cover of the reversal of Σ^k, which is indeed Σ^k itself. The lemma follows by observing that $H^*_0(x) = H^*_0(x^R)$ for any string x. $\square$

The above lemma tells us that if we bound the compression performance of an algorithm in terms of $\overleftarrow{H}^*_k(s)$, then we can achieve the same bound in terms of $H^*_k(s)$ by applying the algorithm to s^R.

For conciseness of exposition in the following sections, we mainly work with $\overleftarrow{H}^*_k(s)$ and $H^*_k(s)$ and we state the results for $\overleftarrow{H}_k(s)$ and $H_k(s)$ in Section 5.1.

2.2. PREFIX CODES AND THEIR LENGTHS. Consider a binary code C for the alphabet Σ and let l_i be the length of the codeword assigned to symbol $a_i \in \Sigma$. The *empirical expected codeword length* for C and string s is

$$L_C(s) = \sum_{i=1}^{h} P_i l_i. \tag{15}$$

An optimal code for the string s is one that minimizes $L_C(s)$ among all possible codes. From now on, we restrict our attention to prefix codes and in particular to Huffman codes, which are optimal. Denote by $L_{HC}(s)$ the empirical expected codeword length of the Huffman code for the string s. Let y be the string obtained by encoding it via a Huffman code. Since in (15), we are using an empirical probability distribution it is $L_{HC}(s) = |y|/|s|$. The following theorem establishes some useful bounds on $L_{HC}(s)$. We omit the proof which can be easily derived from the corresponding results in Capocelli et al. [1986].

THEOREM 2.5. *Let P_1 and P_h be the probabilities of the most likely and least likely symbols in s, with the convention that $P_h = 0$, when s is made of only one symbol. Moreover, let $\beta = 0.0860$, we have*

$$\begin{aligned} L_{HC}(s) &\leq H_0(s) + P_1 - P_h + \beta && \text{if } P_1 < 1/2; && (16)\\ L_{HC}(s) &\leq H_0(s) + 2 - \mathcal{H}(P_1) - P_1 - P_h && && (17)\\ &\leq H_0(s) + P_1 - P_h && \text{if } P_1 \geq 1/2; && (18) \end{aligned}$$

where $\mathcal{H}(P_1)$ denotes the binary empirical entropy of s:

$$\mathcal{H}(P_1) = -P_1 \log P_1 - (1 - P_1)\log(1 - P_1). \tag{19}$$

For the interested reader, we point out that β in (16) comes out from a structural property of an Huffman tree, known as *the sibling property* (see Capocelli et al. [1986] and references therein for details).

```
mississippi$          $ mississipp i
ississippi$m          i $mississip p
ssissippi$mi          i ppi$missis s
sissippi$mis          i ssippi$mis s
issippi$miss          i ssissippi$ m
ssippi$missi    ⟹     m ississippi $
sippi$missis          p i$mississi p
ippi$mississ          p pi$mississ i
ppi$mississi          s ippi$missi s
pi$mississip          s issippi$mi s
i$mississipp          s sippi$miss i
$mississippi          s sissippi$m i
```

FIG. 1. The Burrows–Wheeler transform for the string $s = \texttt{mississippi}$. The matrix on the right has the rows sorted in lexicographic order. The output of the BWT is the last column of the matrix, that is, `ipssm$pissii`.

3. *The Burrows–Wheeler Transform as a Compression Booster*

In 1994, Burrows and Wheeler [1994] introduced a transform that turns out to be very elegant in itself and extremely useful for textual data compression. Given a string s, the transform consists of three basic steps (see Figure 1): (1) append to the end of s a special symbol \$ smaller than any other symbol in Σ; (2) form a *conceptual* matrix $\mathcal{M}$ whose rows are the cyclic shifts of the string $s\$$, sorted in lexicographic order; (3) construct the transformed text $\hat{s} = \texttt{bwt}(s)$ by taking the last column of $\mathcal{M}$. Notice that every column of $\mathcal{M}$, hence also the transformed text $\hat{s}$, is a permutation of $s\$$. Although it is not obvious, from $\hat{s}$ we can always recover s, see Burrows and Wheeler [1994] for details.

Burrows and Wheeler, in their seminal paper, pointed out that the power of the transform seems to rest on the fact that equal contexts (substrings) of s are grouped together resulting in a few clusters of distinct symbols in $\texttt{bwt}(s)$. That clustering makes $\texttt{bwt}(s)$ a better string to compress than s. For a long time, there was only experimental evidence that the transform actually makes compression "easier". Recently, Manzini [2001] has pointed out properties of the transform allowing to state the above experimental evidence in formal terms. We review those insights next.

Let w denote a substring of s. By construction, all the rows of the BWT matrix prefixed by w are consecutive. Hence, the single symbols preceding every occurrence of w in s are grouped together in a set of consecutive positions of the string $\hat{s}$ (last column of $\mathcal{M}$). Then these symbols form a substring of $\hat{s}$, which we denote hereafter by $\hat{s}[w]$. Using the notation introduced in Section 2.1, we have that $\hat{s}[w]$ is equal to a *permutation* of $\overleftarrow{w_s}$, that is, a permutation of the set of single symbols preceding w in s. We write $\hat{s}[w] = \pi_w(\overleftarrow{w_s})$ with π_w being a string permutation which depends on w.

Example 3.1. Let $s = \texttt{mississippi}$ and $w = \texttt{s}$. The four occurrences of s in s are in the last four rows of the BWT matrix. Thus, $\hat{s}[\texttt{s}]$ consists of the last four symbols of $\hat{s}$ and we have $\hat{s}[\texttt{s}] = \texttt{ssii}$. Indeed, those are the single symbols preceding s in s, in a proper order (cfr. $\overleftarrow{w_s} = \texttt{isis}$).

Fix a positive integer k. In the first k columns of the BWT matrix we find, lexicographically ordered, all length-k substrings of s. Hence, the string $\hat{s}$ can be partitioned into the substrings $\hat{s}[w]$ by varying w over Σ^k. We then write

$$\hat{s} = \bigsqcup_{w \in \Sigma^k} \hat{s}[w], \tag{20}$$

where $\bigsqcup$ denotes the concatenation operator among strings.[4] The same argument holds for any prefix cover, and in particular for the prefix cover $\mathcal{P}_k^*$ which defines $\overleftarrow{H}_k^*(s)$: each row of the BWT matrix is prefixed by a unique string in $\mathcal{P}_k^*$ hence

$$\hat{s} = \bigsqcup_{w \in \mathcal{P}_k^*} \hat{s}[w]. \tag{21}$$

This formula drives us to look at the BWT as a tool for reducing the problem of compressing s up to the kth order entropy to the problem of compressing *distinct portions* of $\hat{s}$ up to their *zeroth order* entropy. More precisely, suppose that we have a compression algorithm A that squeezes any string z in $|\mathtt{A}(z)| \leq \lambda |z| H_0^*(z) + \mu$ bits, where λ and μ are constants independent of the input string z. Using A we can compress any string s up to its kth order entropy with the following procedure:

(1) compute $\hat{s} = \mathtt{bwt}(s)$,

(2) find the optimal prefix cover $\mathcal{P}_k^*$, and partition $\hat{s}$ into substrings $\hat{s}[w]$, $w \in \mathcal{P}_k^*$;

(3) compress each substring $\hat{s}[w]$ using algorithm A.

Since $\hat{s}[w]$ is a permutation of $\overleftarrow{w_s}$ we have $|\mathtt{A}(\hat{s}[w])| \leq \lambda |\overleftarrow{w_s}| H_0^*(\overleftarrow{w_s}) + \mu$. By (14), it follows that the above procedure produces an output of size at most $\lambda |s| \overleftarrow{H}_k^*(s) + \mu |\Sigma|^k$ bits. By Lemma 2.4, applying the above procedure to s^R produces an output size of at most $\lambda |s| H_k^*(s) + \mu |\Sigma|^k$ bits.

It goes without saying that, in the above compression procedure, we ignored a few important details such as, the presence of the \$ symbol and the fact that at decompression time we need to be able to detect the end of each substring $\hat{s}[w]$. We will deal with these details when describing our compression booster in Section 5. At this point, however, it is crucial to observe that the above algorithmic approach, although appealing, shows two drawbacks: (1) it needs to compute an optimal prefix cover $\mathcal{P}_k^*$, and (2) its compression can be bounded in terms of a single entropy H_k^*, since the parameter k must be *chosen in advance*.

We overcome both of these drawbacks by making use of the suffix tree data structure [McCreight 1976]. We prove a strong structural relationship between the BWT matrix and the suffix tree of $s\$$. Using this relationship we show that we can find an *absolute optimal* prefix cover whose cost is within a constant factor from the cost of $\mathcal{P}_k^*$ *for any* $k \geq 0$. Therefore, using this absolute optimal prefix cover together with algorithm A, we get an output bounded in terms of the kth order entropy, for any $k \geq 0$. In other words, we have boosted the performance of A from the zeroth order entropy H_0^* to the kth order entropy H_k^*, for any $k \geq 0$. The other

[4] In addition to $\sqcup_{w \in \Sigma^k} \hat{s}[w]$, the string $\hat{s}$ also contains the last k symbols of s (which do not belong to any $\overleftarrow{w_s}$) and the special symbol \$. We momentarily ignore the presence of these $k + 1$ symbols in $\hat{s}$ and deal with them in Section 4.

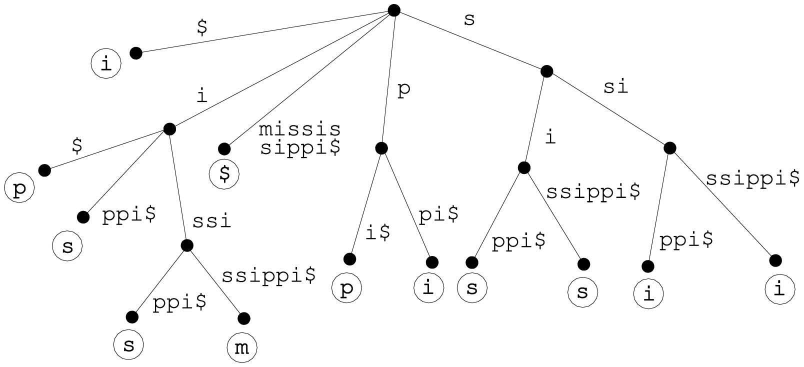

FIG. 2. Suffix tree for the string $s =$ mississippi\$. The symbol associated to each leaf is displayed inside a circle, it is the symbol occupying the corresponding position of $\hat{s}$, when the leaves are listed from left to right.

key contribution of this article is to show that we can find such an absolute optimal prefix cover with a post-order visit of the suffix tree which only takes $O(|s|)$ time.

4. *Prefix Covers, Leaf Covers, Optimal Partitioning*

A crucial ingredient for the design of our compression booster is the relationship between the BWT matrix and the suffix tree data structure. Let $\mathcal{T}$ denote the suffix tree of the string $s\$$. $\mathcal{T}$ has $|s| + 1$ leaves, one per suffix of $s\$$, and edges labelled with substrings of $s\$$ (see Figure 2). Any node u of $\mathcal{T}$ has *implicitly associated* a substring of $s\$$, given by the concatenation of the edge labels on the downward path from the root of $\mathcal{T}$ to u. In that implicit association, the leaves of $\mathcal{T}$ correspond to the suffixes of $s\$$. We assume that the suffix tree edges are sorted lexicographically. As a consequence, if we scan $\mathcal{T}$'s leaves from left to right, the associated suffixes are lexicographically sorted.

Since each row of the BWT matrix is prefixed by one suffix of $s\$$ (see Section 3), there is a natural *one-to-one correspondence* between the leaves of $\mathcal{T}$ and the rows of the BWT matrix. Moreover, since the suffixes are lexicographically sorted, both in $\mathcal{T}$ and in the BWT matrix, the ith leaf (counting from the left) of the suffix tree corresponds to the ith row of the BWT matrix. We associate the ith leaf of $\mathcal{T}$ with the ith symbol of the string $\hat{s}$. The symbol written in the leaf v is thus the symbol preceding in s the substring of $s\$$ associated with v. We write ℓ_i to denote the ith leaf of $\mathcal{T}$ and $\hat{\ell}_i$ to denote its associated symbol. From the above discussion, it follows that $\hat{s} = \hat{\ell}_1 \hat{\ell}_2 \cdots \hat{\ell}_{|s|+1}$. See Figure 2 for an example.

Let w be a substring of s. The *locus* of w is the node $\tau[w]$ of $\mathcal{T}$ that has associated the shortest string prefixed by w. Notice that many strings may have the same locus because a suffix tree edge may be labelled by a long substring of s. For example, in Figure 2, the locus of both ss and ssi is the node reachable by the path labelled by ssi.

4.1. THE NOTION OF LEAF COVER. For any suffix tree node u, let $\hat{s}\langle u\rangle$ denote the substring of $\hat{s}$ concatenating, from left to right, the symbols associated to the leaves descending from node u. For example, in Figure 2, given the node $\tau[\texttt{i}]$ as the locus of the substring i, we have $\hat{s}\langle\tau[\texttt{i}]\rangle =$ pssm. Note that these symbols are exactly the single symbols preceding `i` in `mississippi$`. Indeed, because of the relationship between the suffix tree and the BWT matrix, for any string w we have $\hat{s}\langle\tau[w]\rangle = \hat{s}[w]$.

Given a suffix tree $\mathcal{T}$, we say that a subset $\mathcal{L}$ of its nodes is a *leaf cover* if every leaf of the suffix tree has a *unique* ancestor in $\mathcal{L}$. Any leaf cover $\mathcal{L} = \{u_1, \dots, u_p\}$ naturally induces a partition of the leaves of $\mathcal{T}$ namely $\hat{s}\langle u_1\rangle, \dots, \hat{s}\langle u_p\rangle$. Because of the relationship between $\mathcal{T}$ and the BWT matrix this is also a partition of $\hat{s}$.

Example 4.1. Consider the suffix tree for s = `mississippi` in Figure 2. A leaf cover consists of all nodes of depth one. Using the notion of locus, we can describe this leaf cover as $\mathcal{L}_1 = \{\tau[\texttt{\$}], \tau[\texttt{i}], \tau[\texttt{m}], \tau[\texttt{p}], \tau[\texttt{s}]\}$. Another leaf cover is $\mathcal{L}_2 = \{\tau[\texttt{\$}], \tau[\texttt{i\$}], \tau[\texttt{ip}], \tau[\texttt{is}], \tau[\texttt{m}], \tau[\texttt{p}], \tau[\texttt{si}], \tau[\texttt{ss}]\}$ which is instead formed by nodes at various depths. The partition of $\hat{s}$ induced by $\mathcal{L}_1$ is $\{\texttt{i}, \texttt{pssm}, \$, \texttt{pi}, \texttt{ssii}\}$ and the one induced by $\mathcal{L}_2$ is $\{\texttt{i}, \texttt{p}, \texttt{s}, \texttt{sm}, \$, \texttt{pi}, \texttt{ss}, \texttt{ii}\}$.

We are now ready to describe the relationship between leaf covers and prefix covers. Let $\mathcal{P}_k^* = \{w_1, \dots, w_p\}$ denote an optimal prefix cover which defines $\overleftarrow{H}_k^*(s)$ (see (14)). Let P_k denote the set of nodes $\{\tau[w_1], \dots, \tau[w_p]\}$, which are the loci of the strings in $\mathcal{P}_k^*$. Since $\mathcal{P}_k^*$ is a prefix cover of Σ^k, any leaf of $\mathcal{T}$ corresponding to a suffix of length greater than k has a unique ancestor in P_k. Conversely, a leaf of $\mathcal{T}$ corresponding to a suffix of length shorter than k might not have an ancestor in P_k, because that suffix might be not prefixed by any string in $\mathcal{P}_k^*$. As an example, consider the extreme case in which $\mathcal{P}_k^* = \Sigma^k$; all suffixes of $s\$$ shorter than k are not prefixed by any string in $\mathcal{P}_k^*$ so their leaves have no ancestor in P_k.

This means that in order to transform P_k into a leaf cover for $\mathcal{T}$, we need to add at most k suffix tree nodes. Specifically, let Q_k denote the set of leaves of $\mathcal{T}$ corresponding to suffixes of $s\$$ of length at most k which are not prefixed by a string in $\mathcal{P}_k^*$. We set $\mathcal{L}_k^* = P_k \cup Q_k$ and refer to this set as the leaf cover *associated* to an optimal prefix cover $\mathcal{P}_k^*$. Such a natural relation between leaf and prefix covers is exploited next.

4.2. The Cost of a Leaf Cover. Let C denote the function which associates to every string x over $\Sigma \cup \{\$\}$, with at most one occurrence of \$, the positive real value

$$C(x) = \lambda |x'|\, H_0^*(x') + \mu, \tag{22}$$

where λ and μ are positive constants, and x' is the string x with the symbol \$ removed, if present. The rationale for considering a cost function C of this form is the following. We will use C to measure the compression of the substrings of $\hat{s}$ achieved by a zeroth order encoder (hence, the term H_0^*). Since $\hat{s}$ contains one occurrence of the special symbol \$, a substring of $\hat{s}$ may contain the symbol \$. However, in our algorithm, we store separately the position of \$ within $\hat{s}$ (see Section 5). Therefore, the \$ symbol is ignored in the definition of C. For any leaf cover $\mathcal{L}$, we define its cost as:

$$C(\mathcal{L}) = \sum_{u \in \mathcal{L}} C(\hat{s}\langle u\rangle). \tag{23}$$

Notice that the definition of $C(\mathcal{L})$ is *additive* and, loosely speaking, accounts for the cost of individually compressing the substrings of the partition of $\hat{s}$ induced by $\mathcal{L}$.

Example 4.2. The "smallest" leaf cover of $\mathcal{T}$ is $\{root(\mathcal{T})\}$ and its induced partition consists of the whole string $\hat{s}$. Hence, $C(\{root(\mathcal{T})\}) = C(\hat{s})$. The "largest" leaf cover of $\mathcal{T}$ consists of the suffix tree leaves $\{\ell_1, \ldots, \ell_{|s|+1}\}$ and its induced partition consists of the singletons $\hat{\ell}_1, \ldots, \hat{\ell}_{|s|+1}$. Hence, $C(\{\ell_1, \ldots, \ell_{|s|+1}\}) = \sum_{i=1}^{|s|+1} C(\hat{\ell}_i)$. Note that, according to (22), we have $C(\hat{\ell}_i) = \lambda + \mu$ for $\hat{\ell}_i \in \Sigma$ and $C(\hat{\ell}_i) = \mu$ for $\hat{\ell}_i = \$$.

The next lemma shows that the cost of the leaf cover $\mathcal{L}_k^*$, associated to an optimal prefix cover $\mathcal{P}_k^*$, is within an additive constant from $\lambda|s|\overleftarrow{H}_k^*(s)$.

LEMMA 4.3. *Fix $k \geq 0$. Let C be defined by* (22) *and let $\mathcal{L}_k^*$ be the leaf cover associated with $\mathcal{P}_k^*$. There exists a constant g_k such that, for any string s*

$$C(\mathcal{L}_k^*) \leq \lambda|s|\overleftarrow{H}_k^*(s) + g_k.$$

PROOF. By definition of $\mathcal{L}_k^*$, we have $\mathcal{L}_k^* = P_k \cup Q_k$, hence

$$C(\mathcal{L}_k^*) = \sum_{u \in P_k} C(\hat{s}\langle u \rangle) + \sum_{u \in Q_k} C(\hat{s}\langle u \rangle).$$

To evaluate the second summation, recall that Q_k contains only leaves and that $|Q_k| \leq k$. Moreover, if u is a leaf, then $|\hat{s}\langle u \rangle| = 1$ and $C(\hat{s}\langle u \rangle) \leq \lambda + \mu$. Hence, the second summation is bounded by $k(\lambda + \mu)$. To evaluate the first summation, recall that every $u \in P_k$ is the locus of a string $w \in \mathcal{P}_k^*$. By the relation between the suffix tree and the BWT matrix, we have for any string w that $\hat{s}\langle \tau[w] \rangle = \hat{s}[w]$. Using that and (22), we get

$$\begin{aligned} C(\mathcal{L}_k^*) &\leq \sum_{u \in P_k} C(\hat{s}\langle u \rangle) + k(\lambda + \mu) \\ &= \sum_{w \in \mathcal{P}_k^*} C(\hat{s}[w]) + k(\lambda + \mu) \\ &\leq \lambda \left(\sum_{w \in \mathcal{P}_k^*} |\hat{s}[w]|\ H_0^*(\hat{s}[w]) \right) + \mu|\Sigma|^k + k(\lambda + \mu). \end{aligned}$$

Now recall that $\hat{s}[w]$ is a permutation of $\overleftarrow{w_s}$ and therefore $H_0^*(\hat{s}[w]) = H_0^*(\overleftarrow{w_s})$. Hence, using (14), we get

$$C(\mathcal{L}_k^*) \leq \lambda \left(\sum_{w \in \mathcal{P}_k^*} |\overleftarrow{w_s}|\ H_0^*(\overleftarrow{w_s}) \right) + g_k \ = \ \lambda|s|\overleftarrow{H}_k^*(s) + g_k. \quad \square$$

We now consider a leaf cover $\mathcal{L}_{\min}$ which minimizes $C(\mathcal{L})$, hereafter called an *optimal leaf cover*. That is, $\mathcal{L}_{\min}$ is such that $C(\mathcal{L}_{\min}) \leq C(\mathcal{L})$, for any leaf cover $\mathcal{L}$. We say that $\mathcal{L}_{\min}$ induces an *optimal partition* of $\hat{s}$ with respect to the cost function C. The relevance of $\mathcal{L}_{\min}$ resides in the following lemma, whose proof immediately derives from Lemma 4.3 and the obvious inequality $C(\mathcal{L}_{\min}) \leq C(\mathcal{L}_k^*)$ for any k.

LEMMA 4.4. *Let $\mathcal{L}_{\min}$ be an optimal leaf cover for the cost function C defined by* (22). *For any $k \geq 0$ there exists a constant g_k such that, for any string s*

$$C(\mathcal{L}_{\min}) \leq \lambda|s|\overleftarrow{H}_k^*(s) + g_k.$$

```
(1) Construct the suffix tree $\mathcal{T}$ for the string $s\$$.
(2) Visit $\mathcal{T}$ in post-order. Let $u$ be the currently visited node, and let $u_1, u_2, \ldots, u_c$ be its children, if any:
    (2.1) Compute $C(\hat{s}\langle u \rangle)$.
    (2.2) If $u$ is a leaf set $Z(u) = C(\hat{s}\langle u \rangle)$, otherwise set $Z(u) = \min\{C(\hat{s}\langle u \rangle), \sum_i Z(u_i)\}$.
    (2.3) Set the leaf cover $\mathcal{L}(u) = \{u\}$ if $Z(u) = C(\hat{s}\langle u \rangle)$; otherwise set $\mathcal{L}(u) = \cup_{i=1}^{c} \mathcal{L}(u_i)$.
(3) Set $\mathcal{L}_{\min} = \mathcal{L}(root(T))$.
```

FIG. 3. Pseudocode for the linear-time computation of an optimal leaf cover $\mathcal{L}_{\min}$.

Summing up, we have shown that instead of finding the partition induced by $\mathcal{P}_k^*$, for a fixed k, we can find the optimal partition corresponding to the leaf cover $\mathcal{L}_{\min}$ (this corresponds to what we called the *absolute optimal prefix cover* at the end of Section 3). Such an approach has two remarkable properties. First, the entropy bound of Lemma 4.4 holds *for any k*. Second, $\mathcal{L}_{\min}$ has strong structural properties that allow its computation in optimal linear time. This will be proven in the next section.

4.3. COMPUTING $\mathcal{L}_{\min}$ IN LINEAR TIME.

The key observation for computing $\mathcal{L}_{\min}$ in linear time is a *decomposability property* with respect to the subtrees of the suffix tree $\mathcal{T}$. In the following, with a little abuse of notation, we denote by $\mathcal{L}_{\min}(u)$ an optimal leaf cover of the subtree of $\mathcal{T}$ rooted at the node u.

LEMMA 4.5. *An optimal leaf cover for the subtree rooted at u consists of either the single node u, or of the union of optimal leaf covers of the subtrees rooted at the children of u in $\mathcal{T}$.*

PROOF. When u is a leaf, the result obviously holds, since the optimal leaf cover is the node itself. Assume that u is an internal node and let $u_1, u_2, \ldots, u_c$ be its children in $\mathcal{T}$. Note that both node sets $\{u\}$ and $\cup_{i=1}^{c} \mathcal{L}_{\min}(u_i)$ are leaf covers of the subtree of $\mathcal{T}$ rooted at u. We now show that one of them is optimal for that subtree. Let us assume that $\mathcal{L}_{\min}(u) \neq \{u\}$. Then, $\mathcal{L}_{\min}(u)$ consists of nodes which descend from u. We can then partition $\mathcal{L}_{\min}(u)$ as $\cup_{i=1}^{c} \mathcal{L}(u_i)$, where each $\mathcal{L}(u_i)$ is a leaf cover for the subtree rooted at u_i, child of u in $\mathcal{T}$. By the optimality of the $\mathcal{L}_{\min}(u_i)$'s and the additivity of the function C, we have

$$C(\mathcal{L}_{\min}(u)) = \sum_i C(\mathcal{L}(u_i)) \geq \sum_i C(\mathcal{L}_{\min}(u_i)).$$

Hence, $\cup_i \mathcal{L}_{\min}(u_i)$ is an optimal leaf cover for the subtree rooted at u, as claimed. □

Lemma 4.5 ensures that the computation of $\mathcal{L}_{\min}$ can be done *greedily* by processing bottom-up the nodes of the suffix tree $\mathcal{T}$. The detailed algorithm is given in Figure 3. Note that, during the bottom-up visit of $\mathcal{T}$, we store in $\mathcal{L}(u)$ the optimal leaf cover of the subtree rooted at u and its cost in $Z(u)$.

LEMMA 4.6. *The algorithm in Figure 3 computes the leaf cover $\mathcal{L}_{\min}$ achieving the minimum value for the cost function C defined by (22), in $O(|s|)$ time and $O(|s| \log |s|)$ bits of space.*

PROOF. The correctness of the algorithm follows immediately from Lemma 4.5. For what concerns its running time, the only nontrivial operation during the tree

```
(1) Define the cost function $C$ (see Sect. 4.2) according to the parameters $\lambda$ and $\mu$ which
    appear in the compression bound of algorithm A, as stated by Property 5.1.
(2) Compute an optimal leaf cover $\mathcal{L}_{\min}$ with respect to the cost function $C$.
(3) Compute the partition $\hat{s} = \hat{s}_1 \cdots \hat{s}_m$ induced by $\mathcal{L}_{\min}$.
(4) For $i = 1, \ldots, m$, remove from $\hat{s}_i$ the occurrence of $ (if any) and compress the
    resulting string $\hat{s}'_i$, if not empty, using the algorithm A.
(5) Compress the empty string using algorithm A (A indeed compresses the singleton #).
(6) Write the binary encoding of the position of $ in $\hat{s}$ using $\lfloor \log_2 |s| \rfloor + 1$ bits.
```

FIG. 4. Pseudocode of our compression booster which turns a zeroth order compressor A into a kth order compressor.

visit is the computation of $C(\hat{s}\langle u \rangle)$ in Step (2.1). That requires knowledge of the number of occurrences of the symbol a_i in $\hat{s}\langle u \rangle$, for $i = 1, \ldots, |\Sigma|$. These values can be obtained in $O(1)$ time, as follows: When u is a leaf, then the a_i's occur either zero or one time each. When u is an internal node, the number of occurrences of each a_i can be computed from the number of occurrences of a_i in $\hat{s}\langle u_j \rangle$, where $u_1, \ldots, u_c$ are the children of u in $\mathcal{T}$. (Recall that $|\Sigma| = O(1)$ and $\hat{s}\langle u \rangle$ is the concatenation of $\hat{s}\langle u_1 \rangle, \ldots, \hat{s}\langle u_c \rangle$.) Hence, the visit of the tree takes constant time per node and $O(|s|)$ time overall. Recall that the construction of the suffix tree at Step (1) takes $O(|s|)$ time and $O(|s| \log |s|)$ bits of space. This space bound is also needed to store the values $Z(u)$ during the post-order visit of $\mathcal{T}$. □

5. *A* BWT*-Based Compression Booster*

We are now ready to describe our compression booster that turns a zeroth order compressor into an effective kth order compressor, for any $k \geq 0$, without any (asymptotic) loss in time efficiency. Our starting point is any compressor A satisfying the following property:

PROPERTY 5.1. *Let* A *be a compression algorithm such that, given an input string $x \in \Sigma^*$,* A *first appends an end-of-string symbol # to x and then compresses x# with the following space and time bounds:*

(1) A *compresses x# in at most $\lambda |x| H_0^*(x) + \mu$ bits, where λ and μ are constants,*

(2) *the running time of* A *is $T(|x|)$ and its working space is $S(|x|)$, where $T(\cdot)$ is a convex function and $S(\cdot)$ is nondecreasing.*

Note that instead of using the symbol #, algorithm A can explicitly store the length of x as a prefix of the compressed string x (e.g., using a variable length encoding of the integers among the ones in Elias [1975]); but, in this case, the number of bits used to encode $|x|$ must be accounted for in the output size. An algorithm satisfying Property 5.1 with $\lambda = 2.5$ will be described in Section 6.2.

Given any compressor A satisfying Property 5.1, the algorithmic toolbox detailed in Figure 4 can be used to boost A's compression up to the kth order entropy, for any $k \geq 0$, without any (asymptotic) loss in time efficiency and with a slightly superlinear bit space overhead. The properties of our compression booster are stated in the following theorem.

THEOREM 5.2. *Given a compression algorithm* A *that satisfies Property* 5.1*, the booster detailed in Figure* 4 *yields the following: (A) if applied to s, it compresses it within* $\lambda|s|\,\overleftarrow{H}_k^*(s) + \log_2|s| + g_k$ *bits of storage, for any* $k \geq 0$*; (B) if applied to* s^R*, it compresses s within* $\lambda|s|H_k^*(s) + \log_2|s| + g_k$ *bits, for any* $k \geq 0$. *In both cases, the above compression takes* $O(T(|s|))$ *time and* $O(S(|s|) + |s|\log|s|)$ *bits of space.*

PROOF. We only prove case (A), since (B) would follow from Lemma 2.4. First of all, let us show that the output produced by our booster can be decompressed. Notice that the symbol \$ is initially removed from $\hat{s}$ (i.e., from each $\hat{s}_i$). Hence, each string $\hat{s}'_i$ is over the alphabet Σ. The decoder starts by decompressing the strings $\hat{s}'_1, \ldots, \hat{s}'_m$ one at a time. The end-of-string symbol # is used to distinguish $\hat{s}'_i$ from $\hat{s}'_{i+1}$. Since, at Step (4), we only compress non empty strings $\hat{s}'_i$'s, when the decoder finds an empty string (i.e., the singleton #) it knows that all strings $\hat{s}'_1, \ldots, \hat{s}'_m$ have been decoded and it may compute $|s| = \sum_i |\hat{s}'_i|$. The decoder then fetches the next $(\lfloor \log_2|s| \rfloor + 1)$ bits which contain the position of \$ within $\hat{s}$ (written at Step (6)). At this point, the decoder has reconstructed the original $\hat{s}$ and it may recover the input string s using the inverse BWT.

As far as the compression performance is concerned, we have $|\mathtt{A}(x)| \leq C(x)$ by construction. Since $\mathcal{L}_{\min}$ is an optimal leaf cover with respect to the cost function C, using Lemma 4.4, we get

$$\sum_{i=1}^{m} |\mathtt{A}(\hat{s}_i)| \;\leq\; \sum_{i=1}^{m} C(\hat{s}_i) \;=\; C(\mathcal{L}_{\min}) \;\leq\; \lambda|s|\,\overleftarrow{H}_k^*(s) + g_k.$$

Since the compression of the empty string # needs further μ bits and we append $(\lfloor \log_2|s| \rfloor + 1)$ bits to encode the position of the symbol \$, the overall compression bound follows.

It is simple to prove, for a convex function f and any two positive values a and b, that $f(a) + f(b) \leq f(a+b) + f(0)$. Since T is a convex function and $T(|s|) = \Omega(|s|)$, exploiting the fact that $\sum_i |\hat{s}'_i| = |s|$, we get $\sum_i T(|\hat{s}'_i|) \leq O(T(|s|))$. By Lemma 4.6, we know that computing $\mathcal{L}_{\min}$ takes $O(|s|)$ time. Hence, the overall running time of our booster is $O(T(|s|))$ as claimed. Finally, since space can be reused and S is non-decreasing, the total space used for the compression of $\hat{s}'_1, \ldots, \hat{s}'_m$ is $O(S(|s|))$, and, by Lemma 4.6, computing $\mathcal{L}_{\min}$ takes $O(|s|\log|s|)$ bits. Thus, the overall space occupancy is $O(S(|s|) + |s|\log|s|)$ bits as claimed. □

We point out that, when A admits a decompression algorithm satisfying the same assumptions on time and space as in Property 5.1, the computational resources taken by the decompression process are the same as the ones needed by boosting.

5.1. BOUNDS FOR THE ENTROPY H_k.

The results given for H_k^* can be repeated almost verbatim for the entropy H_k. In this section we state the analogous of Theorem 5.2 for the entropy H_k. The main difference between H_k and H_k^* is that $H_0(a^n) = 0$. Hence, we cannot hope to find an algorithm that satisfies the first requirement of Property 5.1 with H_0^* replaced by H_0. For this reason, we slightly change the space bound for the base algorithm to be used with our compression booster. The new requirements are expressed by the following property:

PROPERTY 5.3. *Let* A *be a compression algorithm such that, given an input string $x \in \Sigma^*$,* A *first appends an end-of-string symbol* # *to* x *and then compresses* x# *with the following space and time bounds:*

(1) A *compresses* x# *in at most* $\lambda|x|H_0(x) + \eta|x| + \mu$ *bits, where* λ, η, *and* μ *are constants,*

(2) *the running time of* A *is* $O(T(|x|))$ *and its working space is* $O(S(|x|))$, *where* $T(\cdot)$ *is a convex function and* $S(\cdot)$ *is non decreasing.*

An algorithm satisfying Property 5.3 with $\lambda = 1$, and $\eta = 1$ will be described in Section 6.1. Given an algorithm A which satisfies Property 5.3, we define, for every string x over $\Sigma \cup \{\$\}$, with at most one occurrence of \$, the cost function:

$$C'(x) = \lambda|x'|\, H_0(x') + \eta|x'| + \mu, \tag{24}$$

where x' is the string x with the symbol \$ removed if present (C' is the analogous of the cost function defined by (22)). Reasoning, as in Section 4, we can define the concept of optimal leaf cover with respect to C' and we can compute it in linear time using the algorithm of Figure 3 with C replaced by C'. Repeating almost verbatim the proof of Theorem 5.2, we get the following result.

THEOREM 5.4. *Given a compression algorithm* A *that satisfies Property* 5.3, *the booster detailed in Figure* 4 *yields the following: (A) if applied to* s, *it compresses it within* $\lambda|s|\, H_k(s) + \log_2|s| + \eta|s| + g_k$ *bits of storage, for any* $k \geq 0$; *(B) if applied to* s^R, *it compresses* s *within* $\lambda|s|H_k(s) + \log_2|s| + \eta|s| + g_k$ *bits, for any* $k \geq 0$. *In both cases, the above compression takes* $O(T(|s|))$ *time and* $O(S(|s|) + |s|\log|s|)$ *bits of space.*

5.2. GENERAL INVERTIBLE AND BOUNDED CONTEXT TRANSFORMS. Let $\mathcal{P}$ be an *invertible string permutation* algorithm. That is, given a string s, $\mathcal{P}$ returns a permutation s_p of the string and, possibly, some additional information of size logarithmic in the length of s that is used for decoding. We say that $\mathcal{P}$ is *realized* by a suffix tree if and only if the following holds for any string s. Let $i_1 i_2 \cdots i_{|s|}$ denote the permutation of the string positions yielding s_p. Then, there exists a permutation π of the nodes in the suffix tree for s such that the leaf associated to the suffix starting at position $i_j + 1$ is the jth leaf in a left to right scan of the leaves of the permuted suffix tree $\pi(\mathcal{T})$. The BWT is certainly realized by a suffix tree. It remains open to establish whether other transforms exist, which substantially depart from sorting permutations. In any case, all the results stated in Section 4 hold for those permutations.

There is one set of permutations that is particularly interesting because of its practical relevance. Fix $k_0 \geq 0$. The *bounded context* transform of order k_0 is the one generated by *sorting* the rows of the BWT matrix according to their prefixes of length k_0 (in the case, of equal prefixes, the rows keep their relative order). The bounded context transforms can be computed more efficiently than the original BWT, and they usually yield only a slightly worse compression. A detailed presentation of those methods as well as of the corresponding decoding algorithms is given in Schindler [1997]. Here we prove that our boosting approach proves helpful in this case too.

THEOREM 5.5. *Let s be a string and s_B be its bounded context transform, for a positive context length k_0. The same results stated in Theorem 5.2 and Theorem 5.4 hold in this case, but only for $k \leq k_0$.*

PROOF. Fix $k \leq k_0$. We give a proof only for $\overleftarrow{H_k^*}(s)$, since the other cases are analogous. The hearth of the proof is to show that the analogous of Lemma 4.3 holds. Now, let $\mathcal{T}_B$ be the suffix tree of s with the leaves rearranged to correspond to the permuted string s_B. That is, the ith leaf of T_B is labelled with the ith symbol of s_B. Notice that the suffix having locus in that leaf is preceded in s by the symbol assigned to the leaf.

We define the *frontier* of depth k_0 in $\mathcal{T}_B$ as the set of nodes such that each one is the locus of a string of length at least k_0 and, among all such nodes, they are the closest to the root. Let $\mathcal{T}'_B$ be the suffix tree $\mathcal{T}_B$, where we have cut all nodes descending from the frontier. So, the nodes on the frontier are now the leaves of $\mathcal{T}'_B$. At each node u of $\mathcal{T}'_B$, either a leaf or an internal node, we assign the string $s_B\langle u\rangle$ obtained by concatenating from left to right the symbols at the leaves of the subtree of $\mathcal{T}_B$ rooted at u. Note that $s_B\langle u\rangle$ is a substring of s_B. Consider a leaf cover $\mathcal{L}_k^*$ for $\mathcal{T}'_B$, associated to $\mathcal{P}_k^*$, and let its cost be defined as in Section 4.2. We can again write $\mathcal{L}_k^* = P_k \cup Q_k$, hence

$$C(\mathcal{L}_k^*) = \sum_{u \in P_k} C(s_B\langle u\rangle) + \sum_{u \in Q_k} C(s_B\langle u\rangle).$$

Reasoning as in the proof of Lemma 4.3 we get $C(\mathcal{L}_k^*) \leq \lambda|s|\overleftarrow{H_k^*}(s) + g'_k$, with the only difference that now such a bound holds only for $k \leq k_0$. Hence, if we compute the optimal leaf cover for $\mathcal{T}'_B$ we are able to compress any string up to $\lambda|s|\overleftarrow{H_k^*}(s) + g'_k$ for any $k \leq k_0$. □

6. *Two Zeroth Order Base Compressors*

In this section, we describe two zeroth order compressors satisfying, with very small constants, the Properties 5.1 and 5.3 stated in Section 5. The first algorithm, called HC, is a variant of the classic Huffman compressor adapted here to efficiently manage the end-of-string symbol #. The second algorithm, called RHC, builds upon HC and uses run-length encoding to efficiently compress the strings with low entropy.

6.1. THE ALGORITHM HC.

In this section, we describe algorithm HC, which satisfies Property 5.3 with $\lambda = 1$ and $\eta = 1$. Let T_1 be a Huffman tree obtained from the empirical probability distribution of s, when s has at least two distinct symbols. Else, it is a tree with only the root, associated to the only symbol in s. Using T_1, we derive a tree T_2 in which there is a codeword also for the symbol #. Indeed, let b be the least frequent symbol in s. T_2 is obtained from T_1 by making the leaf corresponding to b an internal node and by appending to it two new leaves, one assigned to b and the other assigned to #.

Now, the output of algorithm HC consists of an encoding of T_2 followed by the encoding of s# via the codewords represented by T_2.

THEOREM 6.1. *For any string s over the alphabet Σ, we have*

$$|\mathtt{HC}(s\#)| \leq |s|H_0(s) + |s| + \Theta(|\Sigma|\log|\Sigma|). \tag{25}$$

PROOF. We observed that HC(s#) consists of the encoding of T_2 followed by the compressed string y. Let $h = |\Sigma|$. Using for example the algorithm in Witten et al. [1999, Section 2.3] the encoding of T_2 takes $\Theta(h \log h)$ bits. We now show that $|y| \leq |s|H_0(s)+|s|+h+1$. Let $\{P_i\}_{i=1}^{h}$ be the empirical probability distribution of s defined in Section 2.1. The proof proceeds by distinguishing three cases, depending on the value of P_1.

Case $P_1 = 1$. The unique symbol forming s and the symbol # are both encoded by a codeword of one bit. So, $|y| = |s| + 1$ and the bound follows.

Case $1/2 \leq P_1 < 1$. s consists now of more than one symbol. Consider the Huffman tree T_1 obtained from the empirical distribution for s. By construction, the encoding of s via T_1 takes $|s|L_{HC}(s)$ bits (see Section 2.2). Let T_2 be the tree obtained from T_1 as described above and let b denote the least frequent symbol used for its construction. Note that if ℓ_b denotes the length of the codeword representing b in T_1, then $\ell_b + 1$ is the length of the encodings of b and # in T_2. Finally, note that the number of occurrences of b in s is $|s|P_h$. Hence, the encoding of s# via T_2 takes $|s|L_{HC}(s)$ bits plus $|s|P_h$ (one extra bit for each occurrence of b), plus $\ell_b + 1$ (the cost of encoding #). Thus, $|y| = |s|(L_{HC}(s) + P_h) + \ell_b + 1$. Using bound (18) of Theorem 2.5, we get $|y| \leq |s|(H_0(s) + P_1) + \ell_b + 1$. Since T_2 is a binary tree with $h + 1$ leaves, we have $\ell_b \leq h$ and the lemma follows.

Case $P_1 < 1/2$. The proof follows along the same lines as in the previous case, except that now we use bound (16) of Theorem 2.5 and the fact that $|s|(P_1+\beta) < |s|$, since $\beta = 0.0860$ and $P_1 < 1/2$. □

COROLLARY 6.2. *For any string s, if we combine* HC *with the compression booster of Figure* 4 *the resulting algorithm runs in $O(|s|)$ time and $O(|s| \log |s|)$ bits of space, and produces an output bounded by $|s|H_k(s) + |s| + \log |s| + g_k$ bits for any $k \geq 0$.*

6.2. THE ALGORITHM RHC. In this section, we describe algorithm RHC which satisfies Property 5.1 with $\lambda = 2.5$. The main idea behind this algorithm is to use the knowledge about the symbol frequencies in s to make its performance closely approximate H_0^*. As pointed out by Manzini [2001], for low-entropy strings it is essential to use run length encoding. However it is not trivial to combine a zeroth order encoder with run length encoding and to retain "the best of both worlds". Our algorithm here proposes a novel combination, guaranteeing good bounds in terms of $H_0^*(s)$.

Recall that the γ encoding [Elias 1975] of an integer $i \geq 1$ consists of the binary representation i—which takes $1 + \lfloor \log i \rfloor$ bits—prefixed by $\lfloor \log i \rfloor$ 0's. Note that γ codes are prefix-free and that the encoding of i takes $|\gamma(i)| = 1 + 2\lfloor \log i \rfloor$ bits.

Let **0** and **1** be two symbols not in the input alphabet. For any $n \geq 1$, we write $B(n)$ to denote the number n written in binary using **0** and **1** and discarding the most significant bit. For example, $B(1) = \epsilon$ (the empty string), $B(2) = \mathbf{0}$, $B(3) = \mathbf{1}$, $B(4) = \mathbf{00}$, *etc.* Obviously, this encoding is not prefix-free and $|B(i)| = \lfloor \log(i) \rfloor$.

LEMMA 6.3. *Given a sequence of positive integers $d_1, d_2, \ldots, d_t = w$, such that $d_j < d_{j+1}$ for $j = 1, \ldots, t - 1$, there is an algorithm that encodes that* *sequence in at most*

$$|x| \leq \frac{3}{2} t \log\left(\frac{w}{t}\right) + 2t + 2\lfloor \log t \rfloor + 2 \qquad (26)$$

bits. The encoding of the sequence is prefix free in the sense that we can detect the end of the encoding without any additional information.

PROOF. Consider the integer gaps $g_1 = d_1, g_2 = d_2 - d_1, \ldots, g_t = d_t - d_{t-1}$. Our algorithm first encodes the number of gaps t using γ coding, then it encodes $g_1 \sharp g_2 \sharp \ldots g_t \sharp$ where $\sharp$ is a delimiter allowing us to use a variable length (not prefix-free) encoding of each g_j.

The encoding of $g_1 \sharp g_2 \sharp \ldots g_t \sharp$ is a two stage process. In the first stage, we build the string $x' = B(g_1)\sharp B(g_2)\sharp \ldots B(g_t)\sharp$. Let z_0, z_1, and $z_\sharp$ denote respectively the number of $\mathbf{0}$'s, $\mathbf{1}$'s and $\sharp$'s in x'. Assume $z_1 \leq z_0$ (the other case is symmetric: we use one bit to distinguish between the two cases). In the second stage, we construct the binary string x by encoding the symbols of x' with the codewords: $\mathbf{0} \rightarrow 0$, $\mathbf{1} \rightarrow 10$ and $\sharp \rightarrow 11$.

Let $p = \max_i g_i$. For $j = 1, \ldots, p$, let q_j denote the number of g_i's equal to j. Therefore, $\sum_{j=1}^{p} q_j = t$. With reference to the algorithm outlined above, we give a proof only for the case $z_1 \leq z_0$, since the other is symmetric. We have

$$|x| = |\gamma(t)| + z_0 + 2z_1 + 2z_\sharp + 1 = z_0 + 2z_1 + 2t + 2\lfloor \log t \rfloor + 2.$$

We now bound $z_0 + z_1$ in terms of t and $w = d_t$. Since $|B(g_i)| = \lfloor \log(g_i) \rfloor$, we have

$$z_0 + z_1 = \sum_{i=1}^{t} \lfloor \log(g_i) \rfloor = \sum_{j=1}^{p} q_j \lfloor \log(j) \rfloor \leq t \sum_{j=1}^{p} \frac{q_j}{t} \log(j).$$

Using Jensen's inequality and observing that $\sum_{j=1}^{p} j q_j = w$, we get

$$z_0 + z_1 \leq t \log\left(\sum_{j=1}^{p} \frac{q_j}{t} j\right) = t \log\left(\frac{w}{t}\right).$$

Finally, we observe that $z_1 \leq z_0$ implies $z_0 + 2z_1 \leq (3/2)(z_0 + z_1)$ and the lemma follows. □

We point out that the problem considered in Lemma 6.3 can be solved using several known methods for universal encoding of integers [Elias 1975], some of which are asymptotically optimal. However, the asymptotically optimal encodings are better than ours for large integers, while we need to be efficient for all integers, since the lengths of the strings we need to compress (i.e., substrings of $\hat{s}$) may even consist of a few symbols. Therefore, although interesting, asymptotic results do not seem to be applicable here.

In Figure 5 we describe the algorithm RHC. It consists of four cases, based on the empirical probability P_1 of the most frequent symbol occurring in s. Intuitively, as P_1 increases towards one, the empirical entropy of s rapidly decreases and, in order to track it with some degree of accuracy, our algorithm needs to know when to use run length encoding. We remark that the first two bits of the encoding indicate which one among the four cases in the algorithm produces the remaining part of the binary output. We account for those bits in the analysis, but they are not discussed in the presentation of the algorithm. Note that only in the first case we explicitly encode

(a) If $P_1 < 1/2$, we encode $s\#$ using algorithm `HC` (Section 6.1).

(b) If $1/2 \leq P_1 < \frac{|s|-1}{|s|}$, we use run length encoding together with Huffman coding. Let $b = b_1 b_2 \ldots b_r$ be the string resulting from s once that its most frequent symbol a_1 has been removed. We first encode, using Lemma 6.3, the positions where each b_i appears in s and the length $|s|$. Then we encode the string b.
The encoding of b exploits two key facts. First, $P_1 \geq 1/2$ so that the symbol a_1 is associated to a leaf at level one in the Huffman tree T_1 obtained from the empirical probability distribution of s. Second, a_1 does not appear in b so that we can encode b using the tree T_1' obtained by removing from T_1 its root and the leaf for a_1.

(c) If $P_1 = \frac{|s|-1}{|s|}$, s is composed of only two symbols, say a_1 and a_2, and a_2 appears only once. We compress s by concatenating the encoding of the two symbols in the input alphabet, the encoding of $|s|$ given by Lemma 6.3 with $t = 1, w = |s|$, and finally the encoding of the position of a_2 in s using $(\lfloor \log |s| \rfloor + 1)$ bits.

(d) If $P_1 = 1$, s is composed of only one symbol a_1. We output the encoding of a_1 in the input alphabet, and the encoding of $|s|$ given by Lemma 6.3 with $t = 1, w = |s|$.

FIG. 5. Encoding of $s\#$ by algorithm `RHC`.

the symbol #, in all other cases we encode instead the length $|s|$ (see comment immediately following Property 5.1).

The next theorem bounds the output size $|\texttt{RHC}(s\#)|$ in terms of $H_0^*(s)$. We will also be needing the following lemma:

LEMMA 6.4. *Consider a string s and let $r = |s| - n_1$ be the number of symbols remaining once that the most frequent has been deleted. For $1/2 \leq P_1 < 1$, the following bounds holds:*

$$|s|H_0^*(s) = |s|H_0(s) \geq |s|\mathcal{H}(P_1) \tag{27}$$

$$|s|\mathcal{H}(P_1) \geq r[1 + \log(|s|/r)] \tag{28}$$

PROOF. The fact that $\sum_{i=2}^{h} n_i = r$ immediately gives $|s|H_0(s) \geq n_1 \log \frac{|s|}{n_1} + r \log \frac{|s|}{r} = |s|\mathcal{H}(P_1)$. To complete the proof, observe that $n_1 \log \frac{|s|}{n_1} = r \log(1+\frac{r}{n_1})^{\frac{n_1}{r}}$ and that $(1 + 1/t)^t \geq 2$, for $t \geq 1$. □

THEOREM 6.5. *For any string s over the alphabet Σ, we have*

$$|\texttt{RHC}(s\#)| \leq \frac{5}{2}|s|H_0^*(s) + \Theta(|\Sigma| \log |\Sigma|). \tag{29}$$

PROOF. We start by noting that, depending on the various cases, $\texttt{RHC}(s\#)$ is composed of a table, providing a representation of the alphabet symbols as they appear coded in the input string and, possibly, an encoding of a Huffman tree. This part takes $\Theta(|\Sigma| \log |\Sigma|)$ bits [Witten et al. 1999, Sect. 2.3]. Let y be the remaining binary string output by the algorithm, and let $P_1, \ldots, P_h$ be the empirical probability distribution over s. We prove the theorem by showing that $|y| \leq \frac{5}{2}|s|H_0^*(s) + \Theta(|\Sigma|)$.

We distinguish four cases, corresponding to the ones in the algorithm. Note that, except that in case (d), where $P_1 = 1$, we have $H_0^*(s) = H_0(s)$.

(a) $P_1 < 1/2$. In this case, `RHC` produces the same output as algorithm `HC`. Reasoning as in the proof of Theorem 6.1, we have

$$|y| \leq |s|(H_0(s) + P_1 + \beta) + h + 1 \leq |s|H_0(s) + |s| + \Theta(|\Sigma|)\,.$$

The thesis follows observing that $P_1 < 1/2$ implies $|s|\ H_0^*(s) = |s|H_0(s) \geq |s|$.

(b) $1/2 \leq P_1 < (|s| - 1)/|s|$. The output string y now consists of two parts, the first one giving the encoding of $|s|$ and of the position in s of $b_1, \ldots, b_r$, and the second one giving the encoding of b. For the first part, we use Lemma 6.3 with $t = r + 1$, $d_{r+1} = |s|$, and for $i = 1, \ldots, r$, $d_i = p_i$ where p_i is the position of b_i in s. The space required for this encoding is given by (26) with $t = r + 1$ and $w = |s|$.[5]

To bound the length of the encoding of b, we observe that the use of tree T_1' makes that encoding $|s|$ bits shorter than the encoding of s, say y', obtained via the Huffman tree T_1. Putting all of those observations together, we obtain

$$|y| \leq (|y'| - |s|) + \frac{3}{2}(r+1)\log\left(\frac{|s|}{r+1}\right) + 2(r+1) + 2\lfloor\log(r+1)\rfloor + 2. \quad (30)$$

We further bound the right-hand side of (30) by bounding $|y'|$ with (17) of Theorem 2.5 and by using inequality $\log(\frac{|s|}{r+1}) \leq \log(\frac{|s|}{r})$. Applying the fact that $|s|(1 - P_1) = r$ to the result, we get:

$$|y| \leq (|s|H_0(s) - |s|\mathcal{H}(P_1) + r) + \frac{3}{2}(r+1)\log\left(\frac{|s|}{r}\right) + 2r + 2\lfloor\log(r+1)\rfloor + 4. \quad (31)$$

Since $2\lfloor\log(r+1)\rfloor < (r/4) + 5$, inequality (31) simplifies to

$$|y| \leq (|s|H_0(s) - |s|\mathcal{H}(P_1)) + \frac{3}{2}r\log\left(\frac{|s|}{r}\right) + \frac{3}{2}\log\left(\frac{|s|}{r}\right) + (13/4)r + 9. \quad (32)$$

Let $G(r) = r\log\frac{|s|}{r} - \frac{3}{4}r - \frac{3}{2}\log\frac{|s|}{r}$. For $|s| > 16$, $G(2)$ and $G(|s|/2)$ are positive and $G(r)$ is a concave function in $[2, |s|/2]$. Hence, $G(r) > 0$ for $2 \leq r \leq |s|/2$. This implies $\frac{3}{2}\log\frac{|s|}{r} < r\log\frac{|s|}{r} - \frac{3}{4}r$, which plugged into (32) yields

$$|y| \leq (|s|H_0(s) - |s|\mathcal{H}(P_1)) + \frac{5}{2}r + \frac{5}{2}r\log\left(\frac{|s|}{r}\right) + 9. \quad (33)$$

Using now inequality (28), we can bound the right-hand side of (33) to get

$$|y| \leq (|s|H_0(s) - |s|\mathcal{H}(P_1)) + \frac{5}{2}|s|\mathcal{H}(P_1) + 9. \quad (34)$$

Simplifying the right-hand side of (34) and applying (27) to it, the thesis now follows:

(c) $P_1 = \frac{|s|-1}{|s|}$. In this case, y consists of two parts: the encoding of $|s|$ given by Lemma 6.3, which takes $4 + (3/2)(\log|s|)$ bits, followed by $\lceil\log|s|\rceil$ bits. The thesis follows observing that $|s|\ H_0^*(s) = |s|H_0(s) \geq \log|s|$.

(d) $P_1 = 1$. In this case, y only contains the encoding of $|s|$ given by Lemma 6.3 which takes $4 + (3/2)(\log|s|)$ bits. By (9), we have $|s|H_0^*(s) \geq \log|s|$ and the thesis follows.

Finally, we observe that the algorithm should be able to handle also the case $|s| = 0$. This can be easily done using for example one extra bit in case (d). □

[5] If $p_r = |s|$, we cannot use directly Lemma 6.3 since we would have $d_r = d_{r+1}$. In this case, we only encode the sequence $p_1, \ldots, p_r$. This modification does not increase the output size except that we need an additional bit to state whether we are in the case $p_r = |s|$ or not.

Since RHC satisfies Property 5.1, we can transform it into a kth order compressor using the booster described in Section 5.

COROLLARY 6.6. *For any string s, if we combine* RHC *with the compression booster of Figure* 4 *the resulting algorithm runs in $O(|s|)$ time and $O(|s| \log |s|)$ bits of space and produces an output bounded by $2.5|s|H_k^*(s) + \log |s| + g_k$ bits, for any $k \geq 0$.*

7. *Lower Bounds*

In the previous section, we have shown that combining RHC with the compression booster of Section 5 we can compress any string s up to $2.5|s|H_k^*(s) + \log |s| + \Theta(1)$ bits for any $k \geq 0$. In this section, we address the issue of the existence of a zeroth order compressor that satisfies Property 5.1 with $\lambda = 1$. Using such a compressor with our booster we would be able to compress any string up to $|s|H_k^*(s) + \log |s| + \Theta(1)$ for any $k \geq 0$. A related question is whether it is possible to achieve the $|s|H_k^*(s) + \Theta(1)$ bound with a completely different approach, possibly not based on the BWT.

We answer both questions in the negative. Those results come from the stronger finding that, independently of whether the transform is applied as a pre-processing step, the bottleneck for the performance of compression algorithms rests on the need to encode the length of the input string, when it is of the form a^n, that is, $H_0(s) = 0$. We model such a requirement by assuming that a compressor A, when restricted to work on input strings a^n, $n \geq 1$, produces a codeword for n. That is, $\{\mathtt{A}(a^n)|n \geq 1\}$ is a codeword set for the integers. For technical reasons, we also assume that $|\mathtt{A}(a^n)|$ is a nondecreasing function of n. Depending on the implementation, those assumptions seem to account for "the inner working" of many of the best-known compression algorithms that, either implicitly or explicitly, use codes for the integers. Indeed, the decompression algorithm must derive n from the information encoded by the compressor because, since both algorithms must work for any n, they cannot agree once and for all on its value. The lower bound now comes from Theorem 4 in Levenshtein [1968], which we restate in our notation.

THEOREM 7.1 (LEVENSHTEIN 1968, THEOREM 4). *Let* A *be a compressor satisfying the assumptions stated above. Then, there exists a countable number of strings s such that $|\mathtt{A}(s)| \geq |s|H_0^*(s) + \beta(|s|)$, where $\beta(n)$ is a diverging function of n.*

PROOF. Let $s = a^n$. The compressor A, when restricted to work on strings of that form, produces a codeword set for the integers. Then, there must exist infinitely many values of n such that $|\mathtt{A}(s)| \geq \mathrm{logsum}_2 n - (\log_2 \log_2 e) \log_2^* n$, where $\mathrm{logsum}_2 x = \sum_{1 \leq i \leq \log_2^* x} \log_2^{(i)} x$ and $\log_2^* x = \min\{t : \log_2^{(t)} x \leq 1\}$. Since, for $s = a^n$, $|s|H_0^*(s) = 1 + \lfloor \log |s| \rfloor$, we have $|\mathtt{A}(s)| \geq |s|H_0^*(s) + \beta(|s|)$, where $\beta(n)$ is a diverging function of n. □

Combining the above theorem with the observation that $H_k^*(s)$ is a decreasing function of k, we get the following corollary.

COROLLARY 7.2. *No compression algorithm satisfying the assumptions stated at the beginning of the section can compress any string s up to $|s|H_0^*(s)$, even within constant additive factors.*

Note that from Corollary 7.2 follows that we cannot aim at removing both the constant factor 2.5 and the additive term $\log |s|$ from the space bound of Corollary 6.6. Moreover, it settles a conjecture raised in Manzini [2001] stating that no BWT based algorithm can achieve a bound of the form $|s|H_k^*(s) + \Theta(1)$ for any $k \geq 0$.

8. *Conclusions and Open Problems*

We have shown how to boost the performance of compression algorithms. Analytically, our techniques yield compression algorithms that are superior to many of the best-known ones, when one measures performance in a worst case setting, that is, by means of empirical entropy. Moreover, we have given the first family of compression algorithms based on the BWT that do not use MTF as part of the compression process. That settles a long-standing open problem in that area [Fenwick 1996]. We have also addressed the problem of the best possible compression bounds that one can get in terms of empirical entropy of a string, which turns out to be a too demanding bound to achieve.

The contributions in this article raise some interesting open problems. For instance, it would be of great practical interest to conduct an experimental study of our boosting technique. Other issues have a more theoretic flavor, such as to improve the constants of the leading entropy terms in our bounds, to prove tighter lower bounds or investigate the relation among boosting, universal codes of the integers and compression methods such as Huffman codes. Furthermore, our techniques are optimal in time, but use $O(|s| \log |s|)$ bits of working space. It would be therefore interesting to design compression boosters that achieve optimal time and possibly use space (sub)linear in $|s|$. A possible starting point of investigation for this latter issue might be the recent results in Hon et al. [2003]. Finally, we point out that our techniques are off-line in the sense that they look at the whole input before producing any output. It could be worthwhile to investigate the existence of *on-line* data compression boosting techniques.

ACKNOWLEDGMENTS. The authors are deeply indebted to the referees, and in particular one of them, for a very careful reading of the article that lead to very useful, punctual and constructive comments.

REFERENCES

ARNAVUT, Z. 2002. Generalization of the BWT transformation and inversion ranks. In *Proceedings of the IEEE Data Compression Conference*, IEEE Computer Society Press, Los Alamitos, Calif., page 447.

BALKENHOL, B., KURTZ, S., AND SHTARKOV, Y. M. 1999. Modification of the Burrows and Wheeler data compression algorithm. In *Proceedings of the IEEE Data Compression Conference*, IEEE Computer Society Press, Los Alamitos, Calif., page 188.

BENTLEY, J., SLEATOR, D., TARJAN, R., AND WEI, V. 1986. A locally adaptive compression scheme. *Commun. ACM 29*, 4, 320–330.

BURROWS, M., AND WHEELER, D. 1994. A block sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation.

CAPOCELLI, R. M., GIANCARLO, R., AND TANEJA, I. J. 1986. Bounds on the redundancy of Huffman codes. *IEEE Trans. Inf. Theory 32*, 854–857.

CLEARY, J. G., AND TEAHAN, W. J. 1997. Unbounded length contexts for PPM. *Comput. J. 40*, 67–75.

CORMAK, G. V., AND HORSPOOL, R. N. S. 1987. Data compression using dynamic Markov modelling. *Comput. J. 30*, 541–550.

COVER, T. M., AND THOMAS, J. A. 1990. *Elements of Information Theory*. Wiley Interscience, New York.

CROCHEMORE, M., DÉSARMÉNIEN, J., AND PERRIN, D. 2005. A note on the Burrows–Wheeler transform. *Theoret. Comput. Sci. 332*, 567–572.

DEOROWICZ, S. 2002. Second step algorithms in the Burrows–Wheeler compression algorithm. *Softw. Pract. Exper. 32*, 2, 99–111.

EFFROS, M., VISWESWARIAH, K., KULKARNI, S., AND VERDU, S. 2002. Universal codeword sets and representations of the integers. *IEEE Trans. Inf. Theory 48*, 5, 1061–1081.

ELIAS, P. 1975. Universal codeword sets and representations of the integers. *IEEE Trans. Inf. Theory 21*, 2, 194–203.

FENWICK, P. 1996. The Burrows–Wheeler transform for block sorting text compression: principles and improvements. *Comput. J. 39*, 9, 731–740.

GESSEL, I. M., AND REUTENAUER, C. 1993. Counting permutations with given cycle structure and descend set. *J. Combinat. Theory, Ser. A 64*, 189–215.

HON, W., SADAKANE, K., AND SUNG, W. 2003. Breaking a time-and-space barrier in constructing full-text indices. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Los Alamitos, Calif., pp. 251–260.

KARP, R., PIPPENGER, N., AND SIPSER, M. 1985. A Time-Randomness tradeoff. In *Proceedings of the Conference on Probabilistic Computational Complexity*. AMS, 150–159.

KOSARAJU, R., AND MANZINI, G. 1999. Compression of low entropy strings with Lempel–Ziv algorithms. *SIAM Journal on Computing 29*, 3, 893–911.

LARSSON, N. J. 1998. The context trees of block sorting compression. In *Proceedings of the IEEE Data Compression Conference*. IEEE Computer Society Press, Los Alamitos, Calif., pp. 189–198.

LEVENSHTEIN, V. I. 1968. On the redundancy and delay of decodable coding of natural numbers. In (Translation from) *Problems in Cybernetics* (Nauka, Moscow) vol. *20*, 173–179.

MANZINI, G. 2001. An analysis of the Burrows–Wheeler transform. *J. ACM 48*, 3, 407–430.

MCCREIGHT, E. M. 1976. A space-economical suffix tree construction algorithm. *J. ACM 23*, 2, 262–272.

MOFFAT, A. 1990. Implementing the PPM data compression scheme. *IEEE Trans. Commun. 38*, 1917–1921.

NISAN, N., AND TA-SHMA, A. 1999. Extracting randomness: A survey and new constructions. *J. Comput. Syst. Sci. 58*, 148–173.

RISSANEN, J. 1984. Universal coding, information, prediction, and estimation. *IEEE Trans. Inf. Theory IT-29*, 629–636.

SADAKANE, K. 1998. On optimality of variants of the block sorting compression. In *Proceedings of the IEEE Data Compression Conference*. IEEE Computer Society Press, Los Alamitos, Calif., page 570.

SCHAPIRE, R. E. 1990. The strength of weak learnability. *Mach. Learn. 2*, 197–227.

SCHAPIRE, R. E. 2002. The boosting approach to Machine Learning: An overview. In *MSRI Workshop on Nonlinear Estimation and Classification*, D. D. Denison, M. H. Hansen, C. C. Holmes, B. Mallick, and B. Yu, Eds. Springer-Verlag Lecture Notes in Statistics n. 171, 149–172.

SEWARD, J. 1997. The BZIP2 home page. `http://sources.redhat.com/bzip2`.

SCHINDLER, M. 1997. A fast block-sorting algorithm for lossless data compression. In *Proceedings of the IEEE Data Compression Conference*. IEEE Computer Society Press, Los Alamitos, Calif., page 469.

STORER, J. 1992. *Image and Text Compression*. Kluwer Academic Press, Norwell, Mass.

TREVISAN, L. 2001. Extractors and pseudorandom generators. *J. ACM 48*, 860–789.

VALIANT, L. G. 1984. A theory of the Learnable. *Commun. ACM 27*, 1134–1142.

WIRTH, A., AND MOFFAT, A. 2001. Can we do without ranks in Burrows–Wheeler transform compression? In *Proceedings of the IEEE Data Compression Conference*. IEEE Computer Society Press, Los Alamitos, Calif., pp. 419–428.

WITTEN, I. H., MOFFAT, A., AND BELL, T. C. 1999. *Managing Gigabytes: Compressing and Indexing Documents and Images*, Second ed. Morgan Kaufmann Publishers, Los Altos, Calif.

ZIV, J., AND LEMPEL, A. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory 23*, 337–343.

ZIV, J., AND LEMPEL, A. 1978. Compression of individual sequences via variable length coding. *IEEE Trans. Inf. Theory 24*, 530–536.

RECEIVED FEBRUARY 2004; REVISED JANUARY 2005; ACCEPTED FEBRUARY 2005