Università di Pisa
Dipartimento di Informatica

Technical Report: TR-03-10

Optimal compression boosting in optimal linear time using the Burrows-Wheeler Transform.

Paolo Ferragina
Università di Pisa

Giovanni Manzini
Università del Piemonte Orientale

July 3, 2003

ADDRESS: via F. Buonarroti 2, 56127 Pisa, Italy. TEL: +39 050 2212700 FAX: +39 050 2212726

Optimal compression boosting in optimal linear time using the Burrows-Wheeler Transform.*

Paolo Ferragina† — Università di Pisa

Giovanni Manzini‡ — Università del Piemonte Orientale

July 3, 2003

Abstract

In this paper we provide the first compression booster that turns a zeroth order compressor into a more effective k-th order compressor *without any loss in the time and space efficiency.* More precisely, let A be an algorithm that compresses a string s within $\lambda|s|H_0^*(s)+\mu$ bits of storage in $O(T(|s|))$ time, where $H_0^*(s)$ is the zeroth order entropy of the string s. Our booster improves A by compressing s within $\lambda|s|H_k^*(s)+\log_2|s|+g_k$ bits still using $O(T(|s|))$ time, where $H_k^*(s)$ is the k-th order entropy of s.

The idea of a "compression booster" has been very recently introduced by Giancarlo and Sciortino in [6]. They combined the Burrows-Wheeler Transform [3] with dynamic programming and achieved our same compression bound but with running time $O(T(|s|))+\Omega(|s|^2)$. We start from the same premises of [6], but instead of using dynamic programming we design a *linear time* optimization algorithm based on novel structural properties of the Burrows-Wheeler Transform.

1 Introduction

After its proposal in 1994, the Burrows and Wheeler compression algorithm [3] has immediately become a standard for efficient lossless compression. Its key tool is the so called Burrows-Wheeler Transform (BWT, hereafter) that rearranges the symbols of the input string s producing an output $\mathtt{bwt}(s)$ which is easier to compress. Usually, the compression of the string $\mathtt{bwt}(s)$ is implemented via a *cascade* of transformation and coding algorithms: Move-To-Front (MTF), Run-Length Encoding (RLE), and a statistical compressor (Arithmetic or Huffman coding). Even in its simplest forms this scheme achieves a compression performance that is competitive, if not superior, with the best known compression methods (see e.g. [5, 8]). Extensive experimental work has investigated the role and usefulness of each of the above steps, and numerous variants have been proposed in the literature.

Recently, theoretical studies have provided new insights into the nature of the BWT, offering some analytical justifications to the original algorithmic choices made by Burrows and Wheeler in their seminal paper. Specifically, Manzini [7] proved bounds in terms of the k-th order entropy of the string s, denoted by $H_k^*(s)$, *without* making any assumptions on the input s. He showed that a variant of the Burrows-Wheeler compression algorithm takes $(5+\epsilon)|s|H_k^*(s)+\log_2|s|+g_k$ bits to compress any string s, where $\epsilon\approx 10^{-2}$ and g_k is a parameter independent of s. This result is particularly relevant in that the compression bound holds for any $k\geq 0$, and a similar bound cannot hold for many of the best known compression algorithms, like LZ77, LZ78, PPMC.

Given these intriguing properties, more and more researchers have tried to vivisect the structure of the BWT-based compressor and gain insights into the nature and role of its four basic steps: BWT, MTF (Move-to-Front encoding), RLE (Run-length encoding), and zeroth order coding (Huffman or Arithmetic coding). Although MTF and RLE serve their purposes, the general feeling among theoreticians and practitioners was that the MTF coder introduces some degree of inefficiency [1, 2, 4, 9]. Very recently, Giancarlo and Sciortino [6] showed analytically that this feeling was correct. Specifically, they proved that compressing the string s up to its k-th order entropy is

*Partially supported by the Italian MIUR project "Algorithmics for Internet and the Web (ALINWEB)""

†Dipartimento di Informatica, Università di Pisa, Italy. E-mail: ferragina@di.unipi.it

‡Dipartimento di Informatica, Università del Piemonte Orientale, Alessandria, Italy and IIT-CNR, Pisa, Italy. E-mail: manzini@mfn.unipmn.it

equivalent to finding an optimal partition of the string $\mathtt{bwt}(s)$ with respect to a suitably chosen cost function. This idea led to compression algorithms which, in terms of guaranteed performance, outperform the old MTF-based algorithms. For example, in [6] Giancarlo and Sciortino describe a compression algorithm whose output size is bounded by $2.5|s|H_k^*(s) + \log_2|s| + g_k$ bits for any string s and for any $k \geq 0$. This improves by a factor 2 a similar bound for a MTF-based algorithm given in [7].

Giancarlo and Sciortino's approach can be seen as a *compression booster* in that, given a compressor $\mathtt{A}$ which squeezes any string s within $\lambda|s|H_0^*(s) + \mu$ bits of storage, it improves $\mathtt{A}$ by compressing s in $\lambda|s|H_k^*(s) + \log_2|s| + g_k$ bits.[1] A drawback of this booster is that the optimal partition of $\mathtt{bwt}(s)$ is computed using dynamic programming. This induces an overhead of $\Omega(|s|^2)$ time which compares unfavorably with the $\Theta(|s|)$ running time of the MTF-based algorithms.

In this paper we start from the same premises of [6] but, instead of using dynamic programming, we design a *linear time* optimization algorithm based on novel structural properties of the Burrows-Wheeler Transform. Our contribution is twofold. From the theoretical side, we show that the MTF-less compression is overall superior to the classical MTF-based variants. From the practical side, we open the way to novel BWT-based compressors that deserve a careful experimental study.

The bottom line is that we provide compressor designers with the first compression booster that allows to turn a zeroth order compressor into a more effective k-th order compressor, *without any loss in the time and space efficiency.* Their fatiguing task should be simpler, from now on!

2 Notation and known results

In this section we review some known results that will be used in the rest of the paper.

2.1 The empirical entropy of a string

Let s be a string over the alphabet $\Sigma = \{a_1, \ldots, a_h\}$, and let n_i denote the number of occurrences of the symbol a_i inside s. The zeroth order empirical entropy of the string s is defined as

$$H_0(s) = -\sum_{i=1}^{h} \frac{n_i}{|s|} \log\left(\frac{n_i}{|s|}\right) \tag{1}$$

(in the following all logarithms are taken to the base 2 and we assume $0 \log 0 = 0$). The value $|s|H_0(s)$ represents the output size of an ideal compressor which uses $-\log \frac{n_i}{|s|}$ bits for coding the symbol a_i. It is well known that this is the maximum compression achievable by using a uniquely decodable code in which a fixed codeword is assigned to each alphabet symbol.

We can achieve a greater compression if for each symbol we use a codeword which depends on the k symbols preceding it. For any length-k string w we denote by $\vec{w}_s$ the string of symbols following w in s.

Example 1 Let $s = \mathtt{mississippi}$, $w = \mathtt{si}$. The two occurrences of $\mathtt{is}$ inside s are followed by the symbols $\mathtt{s}$ and $\mathtt{p}$, respectively. Hence $\vec{w}_s = \mathtt{sp}$. ∎

The k-th order empirical entropy of s is defined as:

$$H_k(s) = \frac{1}{|s|} \sum_{w \in \Sigma^k} |\vec{w}_s| \, H_0(\vec{w}_s) \tag{2}$$

The value $|s|H_k(s)$ represents a lower bound to the output size of any uniquely decodable encoder which uses for each symbol a code which depends only on the symbol itself and on the k most recently seen symbols. The empirical entropy resembles the entropy defined in the probabilistic setting (for example, when the input comes from a Markov source). However, the 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.

[1] An attentive reader may have noticed the term $\log|s|$ in the compression bounds of [6, 7] which was not present in their published results. 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 [3] and Sect. 2.2). In the papers [6, 7] the compression bounds refer only to the compression of the permutation of s. In the present paper we account also for the space needed to encode the above integer.

In [7] it is shown that for highly compressible strings $|s|H_k(s)$ fails to provide a reasonable bound to the performance of compression algorithms. For this reason it is introduced the concept of *zeroth order modified empirical* entropy as:

$$H_0^*(s) = \begin{cases} 0 & \text{if } |s| = 0 \\ (1 + \lfloor \log |s| \rfloor)/|s| & \text{if } |s| \neq 0 \text{ and } H_0(s) = 0 \\ H_0(s) & \text{otherwise.} \end{cases} \tag{3}$$

Note that for a non-empty 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 k-th order modified empirical entropy H_k^* is 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.[2] 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 Let $\Sigma = \{a, b\}$ and $k = 3$, two examples of suffix covers for Σ^3 are $\{a, b\}$ and $\{a, ab, abb, bbb\}$. Indeed, any string over Σ of length 3 has a unique suffix in both sets. ■

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

$$H_{\mathcal{S}_k}^*(s) = \frac{1}{|s|} \sum_{w \in \mathcal{S}_k} |\vec{w}_s| \, H_0^*(\vec{w}_s); \tag{4}$$

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 *modified k-th order empirical entropy* of s is defined as the compression that we can achieve using the best possible suffix cover. Formally

$$H_k^*(s) = \min_{\mathcal{S}_k \preceq \Sigma^k} H_{\mathcal{S}_k}^*(s), \tag{5}$$

see [7] for further details and properties of the entropy H_k^*. In the following, we use $\mathcal{S}_k^*$ to denote the suffix cover for which the minimum of (5) is achieved. Therefore we write

$$H_k^*(s) = \frac{1}{|s|} \sum_{w \in \mathcal{S}_k^*} |\vec{w}_s| \, H_0^*(\vec{w}_s). \tag{6}$$

A comment is in order at this point. In the rest of the paper we use the Burrows-Wheeler Transform (BWT) as a key component of our compression booster. As we will see, the BWT relates substrings of s with their *preceding* symbols. Conversely, $H_k^*(s)$ relates substrings of s with their *following* symbols. In order to simplify the analysis of our algorithms we introduce an additional notation which offers this other point of view.

Let $\overleftarrow{w}_s$ be the string of symbols that *precede* all occurrences of the substring w in the string s (cfr. $\vec{w}_s$ which contains the following symbols). 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 3 Let $\Sigma = \{a, b\}$ and $k = 3$, two examples of prefix covers for Σ^3 are $\{a, b\}$ and $\{a, ba, bb\}$. Indeed, any string over Σ of length 3 has a unique prefix in both sets. ■

Substituting in (4) $\vec{w}_s$ with $\overleftarrow{w}_s$, and $\mathcal{S}_k$ with $\mathcal{P}_k$, respectively, we define for any prefix cover $\mathcal{P}_k$

$$\overleftarrow{H}_{\mathcal{P}_k}^*(s) = \frac{1}{|s|} \sum_{w \in \mathcal{P}_k} |\overleftarrow{w}_s| \, H_0^*(\overleftarrow{w}_s). \tag{7}$$

In analogy with (5), we take the minimum of (7) over all prefix covers. Let $\mathcal{P}_k^*$ denote the prefix cover which minimizes (7). Using $\mathcal{P}_k^*$, we define the function $\overleftarrow{H}_k^*$ which is analogous to the k-th order empirical entropy (6), but now referring to preceding symbols

$$\overleftarrow{H}_k^*(s) = \frac{1}{|s|} \sum_{w \in \mathcal{P}_k^*} |\overleftarrow{w}_s| \, H_0^*(\overleftarrow{w}_s). \tag{8}$$

The relationship between H_k^* and $\overleftarrow{H}_k^*$ is shown by the following lemma.

[2] Note that H_k was 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 [7] 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`	$\Longrightarrow$	`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`

Figure 1: Example of Burrows-Wheeler transform for the string $s =$ `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; in this example the output is `ipssm$pissii`.

Lemma 1 *For any string s, $\overleftarrow{H}^*_k(s) = H^*_k(s^R)$, where s^R denotes the reversal of the string s.*

Proof: The set of symbols following w in s coincides with the set of 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 Σ^k itself). The lemma follows by observing that $H^*_0(x) = H^*_0(x^R)$ for any string x. ∎

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.

2.2 The Burrows-Wheeler Transform

Let s denote a text over the constant size alphabet Σ. In [3] Burrows and Wheeler introduced a new compression algorithm based on a reversible transformation, now called the *Burrows-Wheeler Transform* (BWT from now on). The BWT consists of three basic steps (see Fig. 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 $\hat{s} = \mathtt{bwt}(s)$ 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 [3] for details.

The importance of the BWT for data compression comes from the following observation. Let w denote a substring of s. By construction, all rows of the BWT matrix prefixed by w are consecutive. Hence, the 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 the previous section, we have that $\hat{s}[w]$ is equal to a *permutation* of $\overleftarrow{w}_s$, namely $\hat{s}[w] = \pi_w(\overleftarrow{w}_s)$ with π_w being a string permutation which depends on w.

Example 4 Let $s =$ `mississippi` and $w =$ `is`. The two occurrences of `is` in s are in the fourth and fifth rows of the BWT matrix. Thus, $\hat{s}[\mathtt{is}]$ consists of the fourth and fifth symbols of $\hat{s}$ and we have $\hat{s}[\mathtt{is}] = \mathtt{ms}$. Indeed, `m` and `s` are the symbols preceding `is` in s. ∎

Note that 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

$$\hat{s} = \bigsqcup_{w \in \Sigma^k} \hat{s}[w] = \bigsqcup_{w \in \Sigma^k} \pi_w(\overleftarrow{w}_s), \tag{9}$$

where $\bigsqcup$ denotes the concatenation operator among strings.[3] 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

[3] 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 Sect. 4.

string in $\mathcal{P}_k^*$ hence

$$\hat{s} = \bigsqcup_{w \in \mathcal{P}_k^*} \hat{s}[w] = \bigsqcup_{w \in \mathcal{P}_k^*} \pi_w(\overleftarrow{w}_s). \tag{10}$$

Recall that permuting a string does not change its zeroth order entropy, that is, $H_0^*(w_s) = H_0^*(\pi_w(w_s))$. Hence, comparing (8) with (10) shows that the BWT can be seen as a tool for reducing the problem of compressing s up to the k-th order entropy to the problem of compressing *distinct portions* of $\hat{s}$ up to their *zeroth order* entropy. Loosely speaking, suppose that we have a compression algorithm $\mathtt{A}$ that squeezes any string z in $|\mathtt{A}(z)| \le \lambda|z|H_0^*(z) + \mu$ bits, where λ and μ are constants independent of the input string z. Using $\mathtt{A}$ we can compress any string s up to $\overleftarrow{H}_k^*(s)$ with the following three-step procedure

1. compute $\hat{s} = \mathtt{bwt}(s)$,
2. find the optimal prefix cover $\mathcal{P}_k^*$, and partition $\hat{s}$ into the substrings $\hat{s}[w]$, $w \in \mathcal{P}_k^*$;
3. compress each substring $\hat{s}[w]$ using algorithm $\mathtt{A}$.

By (8) and (10), we see that the above algorithm produces an output of size at most $\lambda|s|\overleftarrow{H}_k^*(s) + \mu|\Sigma|^k$ bits. By Lemma 1 and by applying the above algorithm to s^R, we get 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 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 4. At this point, however, is crucial to observe that the above algorithmic approach, although appealing, shows two drawbacks: (1) it needs to compute the 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.

In their seminal paper, Burrows and Wheeler [3] implicitly overcame these drawbacks by transforming $\hat{s}$ via the so-called Move-to-Front encoding (MTF from now on) and then compressing the output with an order zero encoder. The analysis in [7] showed that this approach compresses *any string* up to its k-th order entropy, *for any* $k \ge 0$.

Recently, Giancarlo and Sciortino [6] have proposed a new approach for achieving the entropy $H_k^*(s)$ using the BWT. They prove that compressing s up to $H_k^*(s)$, *for any* positive k, is equivalent to finding an optimal partition of the transformed string $\hat{s}$ with respect to a suitably chosen cost function. These ideas lead to compression algorithms which, in terms of guaranteed performance, outperform the old MTF-based approach. A drawback of the Giancarlo and Sciortino's approach is the high computational cost. They use dynamic programming to determine the optimal partition of $\hat{s}$, and this yields an overhead of $\Omega(|s|^2)$ time. This compares unfavorably with the $\Theta(|s|)$ running time of the MTF-based algorithms. In the following sections we show how to take advantage of some structural properties of the BWT matrix in order to find the optimal partition of $\hat{s}$ in $\Theta(|s|)$ time avoiding the use of dynamic programming.

REMARK. In the analysis of BWT-based compressors, it is customary to provide bounds in terms of both the empirical entropies H_k and H_k^* defined in Sect. 2.1 (see [6, 7]). In the following we only consider the entropy H_k^* (which is the hardest to deal with). The analysis for H_k is similar and will be described in the full paper. At the end of Sect. 4 we state, without proof, the main result concerning the entropy H_k.

3 From prefix covers to leaf covers

A crucial ingredient for the efficient computation of the optimal partition of $\hat{s}$ 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 this implicit association the leaves of $\mathcal{T}$ correspond to the suffixes of $s\$$. We assume that the suffix tree edges are ordered lexicographically. As a consequence, if we scan $\mathcal{T}$'s leaves left to right the associated suffixes are lexicographically ordered.

Since each row of the BWT matrix is prefixed by one suffix of $s\$$ (see Section 2.2), there is a natural *one-to-one correspondence* between leaves of $\mathcal{T}$ and rows of the BWT matrix. Moreover, since the suffixes are lexicographically ordered both in $\mathcal{T}$ and in the BWT matrix, the i-th leaf (counting from the left) of the suffix tree corresponds to the i-th row of the BWT matrix. We associate the i-th leaf of $\mathcal{T}$ with the i-th symbol of the string $\hat{s}$. We write ℓ_i to denote the i-th 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.

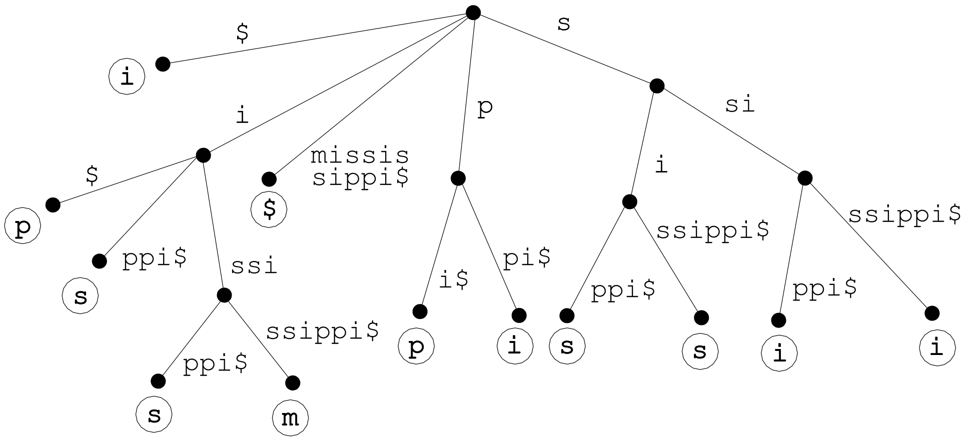

Figure 2: Suffix tree for the string $s = \texttt{mississippi\$}$. The symbol associated to each leaf is displayed inside a circle.

Definition 1 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`.

3.1 The notion of leaf cover

We now introduce the notion of *leaf cover* for T which is related to the concept of prefix cover defined in Sect. 2.1.

Definition 2 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 $\mathcal{L}$. ∎

Example 5 The suffix tree for $s = \texttt{mississippi\$}$ is shown 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 $\mathcal{L}_1 = \{\tau[\$], \tau[\texttt{i}], \tau[\texttt{m}], \tau[\texttt{p}], \tau[\texttt{s}]\}$. Another leaf cover is $\mathcal{L}_2 = \{\tau[\$], \tau[\texttt{i\$}], \tau[\texttt{ippi\$}], \tau[\texttt{issi}], \tau[\texttt{m}], \tau[\texttt{p}], \tau[\texttt{si}], \tau[\texttt{ssi}]\}$ which is instead formed by nodes at various depths. ∎

For any suffix tree node u, let $\hat{s}\langle u\rangle$ denote the substring of $\hat{s}$ containing the symbols associated to the leaves descending from u. For example, in Figure 2 we have $\hat{s}\langle\tau[\texttt{i}]\rangle = \texttt{pssm}$. Note that these symbols are exactly the symbols preceding `i` in `mississippi$`. More in general, we have the following result whose immediate proof follows by the relationship between the suffix tree and the BWT matrix (recall that $\hat{s}[w]$ is the substring of $\hat{s}$ corresponding to the rows prefixed by w, see Sect. 2.2).

Lemma 2 *For any string w, it is $\hat{s}\langle\tau[w]\rangle = \hat{s}[w]$.* ∎

Any leaf cover induces a partition of the suffix tree leaves and therefore a partition of the string $\hat{s}$. Formally, the partition induced by $\mathcal{L} = \{u_1, \ldots, u_h\}$ is $\hat{s}\langle u_1\rangle, \ldots, \hat{s}\langle u_h\rangle$. That this is actually a partition of $\hat{s}$ follows from the definition of leaf cover: since each leaf of $\mathcal{T}$ has an ancestor in $\mathcal{L}$ the concatenation of these substrings covers the whole $\hat{s}$; moreover, these strings are non overlapping, by the "uniqueness" property in Definition 2.

Example 6 Carrying on with Example 5, the partition of $\hat{s}$ induced by $\mathcal{L}_1$ is $\{\texttt{i}, \texttt{pssm}, \$, \texttt{pi}, \texttt{ssii}\}$. The partition of $\hat{s}$ 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 $\mathcal{P}_k^* = \{w_1, \ldots, w_h\}$ denote the optimal prefix cover which defines $\overleftarrow{H}_k^*(s)$ (see (8)). Let P_k denote the set of nodes $\{\tau[w_1], \ldots, \tau[w_h]\}$ which are the loci of the strings in $\mathcal{P}_k^*$. Since $\mathcal{P}_k^*$ is a 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. 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. We formalize this notion with the following definition.

Definition 3 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 $\mathcal{P}_k^*$, and let $\mathcal{L}_k^* = P_k \cup Q_k$. The set $\mathcal{L}_k^*$ is a leaf cover and is called the leaf cover *associated* to the optimal prefix cover $\mathcal{P}_k^*$. ∎

A comment is in order at this point. We are turning prefix covers into leaf covers for the suffix tree $\mathcal{T}$. This effort is motivated by Lemma 2 which shows that there is no difference among different prefixes that have the same locus since they induce the same substring of $\hat{s}$. Intuitively, this means that we can restrict the space in which the optimal prefix cover has to be searched into the one induced by the substrings spelled out by suffix tree nodes. A second crucial observation comes from Definition 2 which constraints the way a subset of these nodes can be selected to form a leaf cover. No every subset of the suffix tree nodes is admissible to form a leaf cover. Finally, Definition 3 characterizes the relation that does exist between leaf covers and prefix covers, thus providing us with a formal bridge between these two concepts, which we will exploit next.

3.2 The cost of a leaf cover

Let C denote the function which associates to every string x over $\Sigma \cup \{\$\}$ the positive real value

$$C(x) = \lambda |x'|\, H_0^*(x') + \mu \tag{11}$$

where λ and μ are positive constants, and x' is the string x with the symbol \$ removed. 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 an 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 4) and for this reason we ignore the symbol \$ in the definition of the cost function C.

Having defined C, for any leaf cover $\mathcal{L}$ we define

$$C(\mathcal{L}) = \sum_{u \in \mathcal{L}} C(\hat{s}\langle u \rangle). \tag{12}$$

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 7 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 $C(\hat{\ell}_i) = \lambda + \mu$ if $\hat{\ell}_i \in \Sigma$, and $C(\hat{\ell}_i) = \mu$ if $\hat{\ell}_i = \$$. ∎

The next lemma shows that the cost of the leaf cover $\mathcal{L}_k^*$ associated to the optimal prefix cover $\mathcal{P}_k^*$ is within an additive constant from the k-th order entropy of s.

Lemma 3 *Let C be defined by* (11) *and let $\mathcal{L}_k^*$ be the leaf cover associated with $\mathcal{P}_k^*$, as defined in Definition 3. For any $k \geq 0$ 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 3 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 $|Q_k| \leq k$. 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 Lemma 2 and (11) we have

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

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 (8)

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

∎

We now consider the leaf cover $\mathcal{L}_{\min}$ which minimizes the value $C(\mathcal{L})$ among all the possible leaf covers $\mathcal{L}$. That is, we are interested in the leaf cover $\mathcal{L}_{\min}$ such that

$$C(\mathcal{L}_{\min}) \leq C(\mathcal{L})$$

for any leaf cover $\mathcal{L}$. We say that $\mathcal{L}_{\min}$ induces the *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 3 and by the trivial observation that $C(\mathcal{L}_{\min}) \leq C(\mathcal{L}_k^*)$.

Lemma 4 *Let $\mathcal{L}_{\min}$ be the optimal leaf cover for the cost function C defined by* (11)*. For any $k \geq 0$ there exists a constant g_k such that, for any string s*

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

∎

Summing up, we have shown that instead of finding the optimal prefix cover $\mathcal{P}_k^*$ we can find the optimal leaf cover $\mathcal{L}_{\min}$. The latter has two remarkable properties: the entropy bound of Lemma 4 holds *for any k*, and $\mathcal{L}_{\min}$ presents strong structural properties that allow its computation in optimal linear time. This is the goal of the next section.

3.3 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 $\mathcal{T}$. With a little abuse of notation, in the following we denote by $\mathcal{L}_{\min}(u)$ the optimal leaf cover of the subtree of $\mathcal{T}$ rooted at node u.

Lemma 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: Let $u_1, u_2, \ldots, u_c$ be the children of u 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 rooted at u (see Definition 2). We now show that one of them is an optimal leaf cover 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 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. ∎

The above lemma ensures that the computation of $\mathcal{L}_{\min}$ admits a *greedy* approach that processes bottom-up the nodes of the suffix tree $\mathcal{T}$. The corresponding algorithm is detailed in Figure 3. Note that during the visit of $\mathcal{T}$ we store in $\mathcal{L}(u)$ the optimal leaf cover of the subtree rooted at u and in $Z(u)$ the cost of such an optimal leaf cover. The correctness of the algorithm follows immediately from Lemma 5.

For what concerns the running time of the algorithm of Fig. 3 we observe that the only non-trivial operation during the tree visit is the computation of $C(\hat{s}\langle u \rangle)$ in Step (2.1). This requires the 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 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. Since the construction of the suffix tree at Step (1) takes $O(|s|)$ time and space, we have established the following result.

Lemma 6 *The algorithm in Figure 3 computes the leaf cover $\mathcal{L}_{\min}$ achieving the minimum value for the cost function C defined by* (11) *in $O(|s|)$ time and space.* ∎

(1) Construct the suffix tree $\mathcal{T}$ for the string $s\$$.
(2) Visit $\mathcal{T}$ in postorder. Let u be the currently visited node, and let $u_1, u_2, \ldots, u_c$ be its children:
    (2.1) Compute $C(\hat{s}\langle u\rangle)$.
    (2.2) Compute $Z(u) = \min\{C(\hat{s}\langle u\rangle), \sum_i Z(u_i)\}$.
    (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 $\mathcal{L}_{\min} = \mathcal{L}(root(T))$.

Figure 3: The pseudocode for the linear-time computation of the optimal leaf cover $\mathcal{L}_{\min}$.

(1) Define the cost function C (see Sect. 3.2) according to the parameters λ and μ which appear in the compression bound of algorithm A.
(2) Compute the 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 actually compresses the singleton #).
(6) Write the binary encoding of the position of \$ in $\hat{s}$ using $\lfloor \log_2 |s| \rfloor + 1$ bits.

Figure 4: The pseudocode of our compression booster which turns a zeroth order compressor A into a k-th order compressor.

4 A BWT-based compression booster

We are now ready to describe our compression booster which turns a zeroth order compressor into a more effective k-th order compressor without any (asymptotic) loss in time and space efficiency. Our starting point is any compressor A which satisfies the following property.

Property 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 $\lambda H_0^*(x) + \mu$ bits, where λ and μ are constant,*
2. *the running time of* A *on input x is $O(T(|x|))$ where $T(\cdot)$ is a convex function.* ■

Note that the algorithm RHC described in [6, Sect. 6] satisfies the above property with $\lambda = 2.5$ and $T(|x|) = |x|$. Given any compressor A satisfying Property 1, the algorithmic toolbox detailed in Figure 4 can be used to boost A's compression up to the k-th order entropy without any (asymptotic) loss in time and space efficiency. The properties of our compression booster are stated in the following theorem.

Theorem 1 *Given a compression algorithm* A *that satisfies Property 1, the booster detailed in Figure 4 compresses any string s within $\lambda |s| \overleftarrow{H}_k^*(s) + \log_2 |s| + g_k$ bits of storage for any $k \geq 0$. The compression takes $O(T(|s|))$ time.*

Proof: 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 (that is, 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, by construction we have $|\mathtt{A}(x)| \leq C(x)$. Since $\mathcal{L}_{\min}$ is the optimal leaf cover with respect to the cost function C, using Lemma 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.

Finally, by the convexity of T and the fact that $\sum_i |\hat{s}'_i| = |s|$ we have $\sum_i T(|\hat{s}'_i|) \leq T(|s|) + O(1)$. By Lemma 6 computing $\mathcal{L}_{\min}$ takes $O(|s|)$ time. Hence, the overall running time of our booster is $O(T(|s|))$ as claimed. ∎

Combining Lemma 1 with Theorem 1, we get that applying our compression booster to the string s^R we can prove the following result.

Corollary 1 *Given a compression algorithm* `A` *that satisfies Property 1 we can compress any string s within $\lambda |s| H_k^*(s) + \log_2 |s| + g_k$ bits for any $k \geq 0$. The above compression takes $O(T(|s|))$ time.* ∎

The analysis that we have carried out for H_k^* can be repeated almost verbatim for the entropy H_k. We will provide the details in the full paper. Here we only state the analogous of Corollary 1 for the entropy H_k.

Corollary 2 *Let* `A` *be a compression algorithm which satisfies Property 1 with the only modification that* `A` *compresses $x\#$ in at most $\lambda H_0(x) + \eta |x| + \mu$ bits, where λ, η and μ are constant. Then, we can compress any string s within $\lambda |s| H_k(s) + \eta |s| + \log_2 |s| + g_k$ bits for any $k \geq 0$. The above compression takes $O(T(|s|))$ time.* ∎

Finally, we observe that our booster does not induce any asymptotic space overhead to the space used by `A`. Indeed, the additional space required by the compression booster of Figure 4 is bounded above by the $O(|s|)$ space required for the construction of the suffix tree.

References

[1] A. Arnavut. Generalization of the BWT trasformation and inversion ranks. In *Proc. IEEE Data Compression Conference*, pages 477–486, 2001.

[2] B. Balkenhol, S. Kurtz, and Y. M. Shtarkov. Modification of the Burrows and Wheeler data compression algorithm. In *Proceedings of IEEE Data Compression Conference*, 1999.

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

[4] S. Deorowicz. Second step algorithms in the Burrows-Wheeler compression algorithm. *Software Practice and Experience*, 32(2):99–111, 2002.

[5] P. Fenwick. The Burrows-Wheeler transform for block sorting text compression: principles and improvements. *The Computer Journal*, 39(9):731–740, 1996.

[6] R. Giancarlo and M. Sciortino. Optimal partitions of strings: A new class of Burrows-Wheeler compression algorithms. In *Combinatorial Pattern Matching Conference (CPM '03)*, pages 129–143, 2003.

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

[8] J. Seward. The BZIP2 home page, 1997. `http://sources.redhat.com/bzip2`.

[9] A.I. Wirth and A. Moffat. Can we do without ranks in Burrows-Wheeler trasform compression ? In *Proc. IEEE Data Compression Conference*, pages 419–428, 2001.