Dipartimento di Informatica
Università del Piemonte Orientale “A. Avogadro”
Via Bellini 25/G, 15100 Alessandria
http://www.di.unipmn.it

The Engineering of a Compression Boosting Library:
Theory vs Practice in BWT compression

Authors: Paolo Ferragina (ferragina@di.unipi.it) Raffaele Giancarlo (raffaele@math.unipa.it)
Giovanni Manzini (manzini@unipmn.it)

TECHNICAL REPORT TR-INF-2006-06-03-UNIPMN
(June 2006)

The University of Piemonte Orientale Department of Computer Science Research Technical Reports are available via WWW at URL `http://www.di.mfn.unipmn.it/`.
Plain-text abstracts organized by year are available in the directory

Recent Titles from the TR-INF-UNIPMN Technical Report Series

2006-02 *A Case-Based Architecture for Temporal Abstraction Configuration and Processing*, Portinale, L., Montani, S., Bottrighi, A., Leonardi, G., Juarez, J., May 2006.

2006-01 *The Draw-Net Modeling System: a framework for the design and the solution of single-formalism and multi-formalism models*, Gribaudo, M., Codetta-Raiteri, D., Franceschinis, G., January 2006.

2005-06 *Compressing and Searching XML Data Via Two Zips*, Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S., December 2005.

2005-05 *Policy Based Anonymous Channel*, Egidi, L., Porcelli, G., November 2005.

2005-04 *An Audio-Video Summarization Scheme Based on Audio and Video Analysis*, Furini, M., Ghini, V., October 2005.

2005-03 *Achieving Self-Healing in Autonomic Software Systems: a case-based reasoning approach*, Anglano, C., Montani, S., October 2005.

2005-02 *DBNet, a tool to convert Dynamic Fault Trees to Dynamic Bayesian Networks*, Montani, S., Portinale, L., Bobbio, A., Varesio, M., Codetta-Raiteri, D., August 2005.

2005-01 *Bayesan Networks in Reliability*, Langseth, H., Portinale, L., April 2005.

2004-08 *Modelling a Secure Agent with Team Automata*, Egidi, L., Petrocchi, M., July 2004.

2004-07 *Making CORBA fault-tolerant*, Codetta Raiteri D., April 2004.

2004-06 *Orthogonal operators for user-defined symbolic periodicities*, Egidi, L., Terenziani, P., April 2004.

2004-05 *RHENE: A Case Retrieval System for Hemodialysis Cases with Dynamically Monitored Parameters*, Montani, S., Portinale, L., Bellazzi, R., Leonardi, G., March 2004.

2004-04 *Dynamic Bayesian Networks for Modeling Advanced Fault Tree Features in Dependability Analysis*, Montani, S., Portinale, L., Bobbio, A., March 2004.

2004-03 *Two space saving tricks for linear time LCP computation*, Manzini, G., February 2004.

2004-01 *Grid Scheduling and Economic Models*, Canonico, M., January 2004.

2003-08 *Multi-modal Diagnosis Combining Case-Based and Model Based Reasoning: a Formal and Experimental Analisys*, Portinale, L., Torasso, P., Magro, D., December 2003.

2003-07 *Fault Tolerance in Grid Environment*, Canonico, M., December 2003.

The Engineering of a Compression Boosting Library: Theory vs Practice in BWT compression

Paolo Ferragina* Raffaele Giancarlo† Giovanni Manzini‡

Abstract

Data Compression is one of the most challenging arenas both for algorithm design and engineering. This is particularly true for Burrows and Wheeler Compression a technique that is important in itself and for the design of compressed indexes [20]. There has been considerable debate on how to design and engineer compression algorithms based on the BWT paradigm. In particular, Move-to-Front Encoding is generally believed to be an "inefficient" part of the Burrows-Wheeler compression process. However, only recently two theoretically superior alternatives to Move-to-Front have been proposed, namely Compression Boosting [7] and Wavelet Trees [6, 10]. The main contribution of this paper is to provide the first experimental comparison of these three techniques, giving a much needed methodological contribution to the current debate. We do so by providing the first carefully engineered compression boosting library that can be used, on the one hand, to investigate the myriad new compression algorithms that can be based on boosting, and on the other hand, to make the first experimental assessment of how Move-to-Front behaves with respect to its recently proposed competitors. The main conclusion is that Boosting, Wavelet Trees and Move-to-Front yield quite close compression performance. Finally, our extensive experimental study of boosting technique brings to light a new fact overlooked in 10 years of experiments in the area: a fast adapting order-zero compressor is enough to provide state of the art BWT compression by simply compressing the run length encoded transform: Move-to-Front, Wavelet Trees, and Boosters can all be by-passed by a fast learner.

1 Introduction

In the quest for the ultimate data compressor, Algorithmic Theory and Engineering go hand in hand. This point is well illustrated by the amount of results and implementations originated by the fundamental results by Lempel and Ziv. A more recent example is provided by the fundamental contributions given by Burrows and Wheeler to data compression [3], via their transform (denoted for short bwt). In their seminal paper Burrows and Wheeler proposed to compress the output of the bwt using Move-to-Front Encoding (shortly mtf), followed by an order zero compressor A (usually Arithmetic or Huffman coding). As pointed out by Fenwick [5] in the first systematic study of that new type of compression, the technique is so powerful that it yields nearly state-of-the-art compression results without any particularly sophisticated engineering of the coding step. This should be contrasted with PPM-based compressors that involve quite a bit of engineering. From that point on, the research on bwt compression has focused on two aspects:

*Dipartimento di Informatica, University of Pisa, Italy. ferragina@di.unipi.it. Partially supported by Italian MIUR Italy-Israel FIRB Project "Pattern Discovery Algorithms in Discrete Structures, with Applications to Bioinformatics"; and by Yahoo! Research grant on "Data compression and indexing in hierarchical memories".

†Dipartimento di Matematica ed Applicazioni, Università di Palermo, Italy. raffaele@math.unipa.it. Partially supported by Italian MIUR grants PRIN "Metodi Combinatori ed Algoritmici per la Scoperta di Patterns in Biosequenze" and FIRB "Bioinformatica per la Genomica e La Proteomica" and Italy-Israel FIRB Project "Pattern Discovery Algorithms in Discrete Structures, with Applications to Bioinformatics".

‡Dipartimento di Informatica, University of Piemonte Orientale, Italy. manzini@mfn.unipmn.it. Partially supported by Italian MIUR Italy-Israel FIRB Project "Pattern Discovery Algorithms in Discrete Structures, with Applications to Bioinformatics".

faster bwt computation, and the identification and exploitation of potential inefficiencies in the use of mtf. While substantial progress has been made on the first point, both theoretically and experimentally (e.g. [2, 18]), the second point experienced a plethora of heuristically-designed proposals (see [1, 4] and references therein) which improved over the original proposal but often lacked of analytical justification.

Recently, two theoretical results [6, 7] have shed new light on the role of mtf within the bwt-based compression paradigm, paving the way to the (*analytically justified*) design of more powerful bwt-based compressors. In particular, [7] proposed a new technique, named *compression boosting*, that fully uses the power of bwt to show that the performance of *any* order zero compressor A can be automatically, and optimally, boosted to higher order entropy compression. On the other hand, [6] proved that combining the bwt with the Wavelet Tree data structure [10] we can achieve high-order entropy bounds without using mtf or the boosting technique. At the same time, a novel and very recent analysis of classic bwt compression [12] showed that mtf may not be as inefficient as initially thought. Summing this with the fact that the theoretical results in [6, 7] require some sophisticated algorithmic machinery, it is not at all clear how much computational/compression *gain* can be achieved by shaving off the mtf-step from the bwt-based compressors.

The above is the main question addressed in the present paper, whose key contribution is first of all *methodological*. We provide the first carefully engineered compression boosting library that can be used, on the one hand, to investigate the myriad new compression algorithms that can be based on boosting, and on the other hand, to make the first experimental assessment of how mtf behaves with respect to its recently proposed competitors: Boosting and Wavelet Trees. The boosting library is available under the GPL license at the page http://www.mfn.unipmn.it/~manzini/boosting and it is highly modular in the sense that it can be used to create a powerful high order compressor even without any knowledge of the bwt.

In order to highlight our additional technical contributions, we need to recall a few facts about compression boosting [7]. Additional details are given in Section 3. The boosting technique builds upon three main ingredients: bwt, the Suffix Tree data structure, and a greedy algorithm to process them. Specifically, it is shown that there exists a proper partition of the bwt of a string s exhibiting a deep combinatorial relation with the k-th order entropy of s. That partition can be identified via a greedy processing of the suffix tree of s. The final compressed string is then obtained by compressing individually each substring of the partition by means of the base (order zero) compressor A we wish to boost. The proper design of a compression booster is a bit trickier than it sounds:

(**A**) The greedy algorithm alluded to before is a bottom up visit of the suffix tree. In practice, on large files, the memory requirements for the construction of the suffix tree would be prohibitively large. We use suffix arrays instead and procedures that efficiently simulate the bottom up visit of the suffix tree [13].

(**B**) Given the algorithm A we wish to boost, we also need an objective function that estimates how well A compresses a given string. In [7], the objective function is given in terms of two parameters λ and μ and the order zero empirical entropy of the string (see Section 3 for details). In practice, λ and μ may either be not available or be too conservative. This point is discussed in Section 4, where we propose two cost models and the relative objective functions.

(**C**) Another important aspect of the boosting process is the ability of the algorithm A to quickly adapt to the statistics of a string to be compressed. Faster adaptation means better compression. This learning process is usually governed by parameters establishing how fast A "forgets the past". We limit our experimentation to range coding and arithmetic coding. The somewhat intuitive, yet surprising, results are reported in Section 5 and briefly outlined in point (**F**) below.

Using our library we have compared the performance of the compression booster against bwt compressors based on mtf (e.g. Bzip2 [21] and variants), bwt compressors based on Wavelet Trees (e.g., Wzip [9]), and state-of-the-art PPM compressors (e.g. PPMd [23]). We show that:

(**D**) As predicted by Theory [7], boosting is superior to classic bwt approaches that use mtf in terms of compression ratio but not by much. It is also slower, as it is to be expected, because of the significant time cost for building the optimal bwt-partition (as observed in **B**). Therefore, those results give a strong indication that mtf may actually be a time-efficient way to effectively "approximate" the optimal partition computed by the boosting technique.

(**E**) As predicted by Theory [6, 10, 11], the simple combination of bwt with Wavelet Trees is very effective both in time and compression ratio and does not benefit from the use of the booster. However, the Wavelet Tree approach is outperformed by classic bwt approaches that use mtf. This further confirms the effectiveness in time and compression ratio of mtf, and leaves open the problem of investigating the more powerful approach proposed in [6], namely *Generalized Wavelet Trees*, which are based on sophisticated combinations of binary (like, Run Length encoders) versus non-binary (like, Huffman or Arithmetic encoders) compressors and Wavelet Trees of properly-designed shapes.

(**F**) The experiments performed to estimate the best adaptation parameters for range and arithmetic coding show clearly that a fast adaptation yields state-of-the-art compression by simply compressing a run length encoded bwt. This is somewhat intuitive, yet surprising: to our knowledge no one observed experimentally the superiority of this strategy w.r.t. mtf, and no theoretical analysis has explained or suggested such behavior. Moreover, this result comes from the stronger finding that for a fast adapting range coder the optimal partition coming out of the booster is the bwt itself (data not shown, due to space limitations). That is, the strategy is optimal.

(**G**) All the bwt-based compressors we tested were inferior, in terms of compression ratio, to the highly engineered PPMd tool. The principle behind bwt and PPM techniques is the same: discover and encode according to the "best" contexts. However, bwt-based algorithms have the advantage of knowing the entire string, while PPMd "discovers" good contexts on-line. Yet bwt-based algorithms do not perform as well. This yields an extremely intriguing engineering problem for data compression practitioners. Note that there is a very good reason to stick with bwt-based compressors instead of embracing the, apparently superior, PPM-based compressors: the reason is that bwt-based compressors are a key tool for the construction of compressed indices which (informally) are compressed files offering the additional capability of very fast full text search (see [20] for formal definitions and a comprehensive survey).

In conclusion our experiments show that Boosting, Wavelet Trees and mtf yield quite close compression performance. However, the boosting technique appears to be more robust and works well even with less effective order zero compressors (such as Huffman coding). Moreover, when used with range/arithmetic coding the boosting technique yields excellent compression somewhat irrespective of how fast the order-zero compressor adapts to the statistics of the string. These positive features are achieved using more resources (time and space) during compression: nevertheless our results show that a careful implementation of boosting can handle efficiently even very large files.

2 Background and Notation

Let s be a string over the alphabet $\Sigma = \{a_1, \dots, a_h\}$ and, for each $a_i \in \Sigma$, let n_i be the number of occurrences of a_i in s. The 0*-th order empirical entropy* of the string s is defined as[1] $H_0(s) = -\sum_{i=1}^{h}(n_i/|s|)\log(n_i/|s|)$. It is well known that H_0 is the maximum compression we can achieve using a fixed codeword for each alphabet symbol. We can achieve a greater compression if the codeword we use for each symbol depends on the k symbols preceding it, since the maximum compression is now bounded by the k-th order entropy $H_k(s)$ (see [15] for the formal definition). For highly compressible strings, $|s|\,H_k(s)$ fails to provide a reasonable bound to the performance of compression algorithms (see discussion in [7, 15]). For that reason, [15] introduced the notion of 0*-th order modified empirical* entropy:

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

[1]We assume that all logarithms are taken to the base 2 and $0 \log 0 = 0$.

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

Figure 1: The 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, i.e., ipssm$pissii.

Note that if $|s| > 0$, $|s|H_0^*(s)$ 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 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.

Given a string s, the Burrows-Wheeler transform (bwt for short) 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 $\hat{s} = \texttt{bwt}(s)$ 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 [3] for details. The power of the bwt rests 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. In their seminal paper Burrows and Wheeler proposed to compress the output of the bwt using Move-to-Front Encoding (shortly mtf), followed by an order zero compressor (Arithmetic or Huffman coding). In [12] it is shown that if we use an order zero compressor A such that for any string x we have $|\mathtt{A}(x)| \le |x|H_0(x) + c|x|$, then the combination: bwt followed by mtf, followed by A produces an output bounded by

$$\mu|s|H_k(s) + (\log\zeta(t) + c)|s| + \log|s| + \mu g_k \tag{2}$$

where ζ is the Riemann zeta function. The above bound holds for any $k \ge 0$ and $t > 1$. Concerning H_k^*, in [15] it is shown that if we use Run Length Encoding (shortly rle) between mtf and the order zero compressor, the output is bounded by

$$(5 + \epsilon)|s|H_k^*(s) + \log_2|s| + g_k' \tag{3}$$

for any $k \ge 0$ and $\epsilon \approx 10^{-2}$. The bottom line is that combining the Burrows-Wheeler transform with mtf and an order zero compressor we can achieve the k-th order entropy, H_k or H_k^*, simultaneously for any $k \ge 0$. Note however, that the coefficient in front of the k-th order entropy in (2) and (3) is greater than 1 whereas we are assuming that A achieves H_0 without any multiplicative constant. This means that there is a small inefficiency as we go from H_0 and H_0^* to H_k and H_k^*. It is an open question wether this inefficiency can be removed with a more detailed analysis or is inherent in the use of Move-to-Front encoding. We point out that other techniques for achieving the k-th order entropy via the bwt have been proposed in the compressed index literature (see [20] for a general survey). In this paper we consider only "pure" compressors which are more space efficient since they do not need the extra information required for supporting indexing operations.

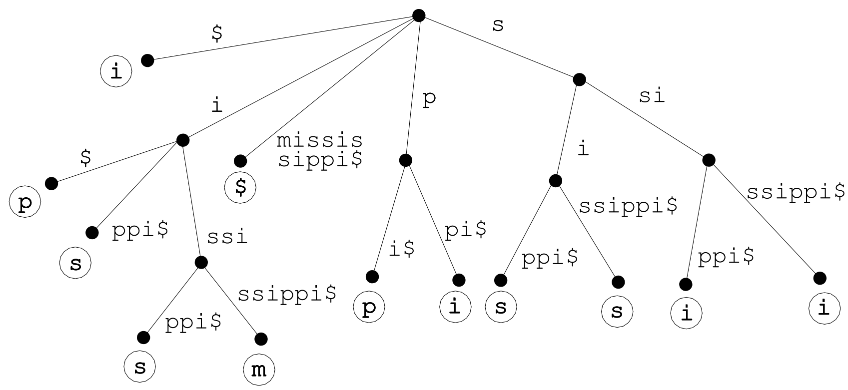

Figure 2: Suffix tree for the string $s =$ `mississippi$`. The symbol associated to each leaf is displayed inside a circle, it is the symbol occupying the corresponding position of $\hat{s}$, when the leaves are listed from left to right.

3 A BWT-based compression booster

Recently, [7] has described a `bwt`-based compression booster that, starting from an order zero compressor, achieves the k-th order entropy without the inefficiency found in the `mtf`-based approach. In this section we quickly review how the boosting algorithm works; the details and proofs can be found in [7].

A crucial ingredient of the 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 2), 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 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 $\hat{s} = \mathtt{bwt}(s)$. 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 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.

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. 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 $\tau[\mathtt{i}]$ as the locus of the substring `i`, we have $\hat{s}\langle\tau[\mathtt{i}]\rangle = \mathtt{pssm}$. 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 $\mathcal{L} = \{u_1, \ldots, u_p\}$ naturally induces a partition of the leaves of $\mathcal{T}$ namely $\hat{s}\langle u_1\rangle, \ldots, \hat{s}\langle u_p\rangle$. Because of the relationship between $\mathcal{T}$ and the `bwt` matrix this is also a partition of $\hat{s}$.

Example 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 $\mathcal{L}_1 = \{\tau[\$], \tau[\mathtt{i}], \tau[\mathtt{m}], \tau[\mathtt{p}], \tau[\mathtt{s}]\}$. Another leaf cover is $\mathcal{L}_2 = \{\tau[\$], \tau[\mathtt{i\$}], \tau[\mathtt{ip}], \tau[\mathtt{is}], \tau[\mathtt{m}], \tau[\mathtt{p}], \tau[\mathtt{si}], \tau[\mathtt{ss}]\}$ which is instead formed by nodes at various depths. The partition of $\hat{s}$ induced by $\mathcal{L}_1$ is $\{\mathtt{i}, \mathtt{pssm}, \$, \mathtt{pi}, \mathtt{ssii}\}$ and the one induced by $\mathcal{L}_2$ is $\{\mathtt{i}, \mathtt{p}, \mathtt{s}, \mathtt{sm}, \$, \mathtt{pi}, \mathtt{ss}, \mathtt{ii}\}$. ■

(1) Construct the suffix tree $\mathcal{T}$ for the string $s\$$.
(2) Visit $\mathcal{T}$ in postorder. Let u be the visited node, and let $u_1, \ldots, u_c$ 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 an optimal leaf cover $\mathcal{L}_{\min}$.

Let C denote a function which associates to every string x over $\Sigma \cup \{\$\}$ the positive real value $C(x)$. For any leaf cover $\mathcal{L}$, we define its cost as:

$$C(\mathcal{L}) = \sum_{u \in \mathcal{L}} C(\hat{s}\langle u \rangle). \tag{4}$$

Example 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 consists of the tree leaves $\{\ell_1, \ldots, \ell_{|s|+1}\}$. Its induced partition consists of the singletons $\hat{\ell}_1, \ldots, \hat{\ell}_{|s|+1}$, hence $C(\{\ell_1, \ldots, \ell_{|s|+1}\}) = \sum_{i=1}^{|s|+1} C(\hat{\ell}_i)$. ■

In [7] it is proven that the procedure in Figure 3 computes a leaf cover $\mathcal{L}_{\min}$ of minimum cost. That is, $\mathcal{L}_{\min}$ is such that $C(\mathcal{L}_{\min}) \leq C(\mathcal{L})$, for any leaf cover $\mathcal{L}$. $\mathcal{L}_{\min}$ is called an *optimal leaf cover* and we say that $\mathcal{L}_{\min}$ induces an *optimal partition* of $\hat{s}$ with respect to the cost function C. The relevance of $\mathcal{L}_{\min}$ for achieving the k-th order entropy derives by the following Theorem proven in [7].

Theorem 1 *Let* $\mathtt{A}$ *denote an order zero compressor such that for any string* x *we have* $|\mathtt{A}(x)| \leq \lambda |x| H_0^*(x) + \mu$ *where* λ *and* μ *are constants. Let* $\mathcal{L}_{\min}$ *denote an optimal partition of* $\hat{s}$ *with respect to the cost function* $C(x) = \lambda |x|\, H_0^*(x) + \mu$. *Then, if we use algorithm* $\mathtt{A}$ *to compress the substrings of the optimal partition of* $\hat{s}$ *induced by* $\mathcal{L}_{\min}$, *the overall output size is bounded by* $\lambda |s|\, H_k^*(s) + g_k$ *bits for any* $k \geq 0$. ■

A similar result holds for the entropy H_k as well: if the output of $\mathtt{A}$ is bounded in terms of H_0, we can use the booster to achieve the k-th order entropy H_k for any $k \geq 0$ (see [7] for details). Since the algorithm of Fig. 3 consists of a postorder visit of the suffix tree $\mathcal{T}$, and the visit of each node takes constant time, the computation of the optimal partitioning takes $O(|s|)$ time.

4 The compression boosting library

The efficient implementation of the compression booster algorithm is a non trivial engineering task. The main challenge is avoiding the explicit construction of the suffix tree which would require an unpractically large amount of working memory.

We now detail our implementation discussing its space requirements in the "real world" model where we assume that every character takes one byte and every integer takes 4 bytes. Let $n = |s|$. We first compute the suffix array of s using the `ds` algorithm [18] that has a peak memory usage of only $5.03n$ bytes: n bytes for the text, $4n$ for the suffix array, and $0.03n$ working space. The algorithm `ds` is one of the fastest available algorithms for "real world" files but its worst-case running time is $\Theta(n^2 \log n)$. If one desires a space economical algorithm with a smaller worst-case running time the best alternative is an algorithm by Burkhardt and Kärkkäinen [2] which runs in $O(n \log n)$ time and uses $5n + O(n/\log n)$ space.

Given the suffix array we compute and store $\hat{s} = \mathtt{bwt}(s)$ using n bytes. To compute the optimal partition of $\hat{s}$ avoiding the explicit construction of the suffix tree we use the technique from [13] that allows one to emulate the post-order visit of the suffix tree using the Longest Common Prefix (shortly LCP) array. Thus, we use the Lcp6 algorithm from [16] for computing in $O(n)$ time the LCP array given s, $\hat{s}$, and the suffix array. This algorithm overwrites the LCP array over the suffix array and has a peak space usage of $(6+\delta)n$ bytes. The parameter δ is at most 4 and is bounded also by $|\Sigma|^k/n + 2H_k(s)$ for any $k \geq 0$. This means that the space usage is smaller for highly compressible inputs (see [16] for details).

Having computed the LCP array we can discard the input string s; thus at this stage wa are only storing $\hat{s}$ and the LCP array for a total space usage of $5n$ bytes. The computation of the optimal partition using the algorithm of Fig. 3 and the technique in [13] reduces to a left to right scan of the LCP array. This allows us to store the endpoints of intervals of the optimal partition in the same memory used for the LCP array (that is, overwriting the LCP array). Thus the only additional memory used during the "emulated" suffix tree visit is the space used to store the stack of the suffix tree nodes whose visit has started but not yet finished. This space could be $\Theta(n)$ in the worst case, but in practice is much smaller than n bytes overall (see Figure 6).

Cost models. An important issue in the implementation of the compression booster is the choice of the parameters λ and μ in the cost function $C(x) = \lambda|x|H_0^*(x)+\mu$ of Theorem 1. Given a compressor $\mathtt{A}$, theory dictates that λ and μ be chosen so that $|\mathtt{A}(x)| \leq C(x)$ for any string x. However, if we strictly enforce this condition it is possible that for many strings x we have $C(x) \gg |\mathtt{A}(x)|$. Since the optimal partitioning is computed minimizing $C(\mathcal{L}_{\min})$, if $C(x)$ is "too far" from $|\mathtt{A}(x)|$ we could end up with a partition which do not exploit the full potential of the compressor $\mathtt{A}$. To evaluate this phenomenon our boosting library supports two different cost models. In addition to the "entropy bound" model outlined above, we provide a "real cost" model in which the optimal partition is computed with respect to the cost $C(x) = |\mathtt{A}(x)|$. Using the "real cost" model we get the best possible compression that we can achieve using the compressor $\mathtt{A}$. The drawback of this model is that during the suffix tree visit the processing of a node no longer takes constant time and therefore the whole procedure no longer runs in linear time. The time cost might be quadratic in the worst case, although the experimental results show that the overall running time usually increases only by a factor 1.5 (see Figures 5 and 6)

For implementing the "entropy bound" model we used the following approach. Instead of determining the parameters λ and μ so that $|\mathtt{A}(x)| \leq C(x)$ for any string x, we use a cost function of the form

$$C(x) = |x|\ H_0^*(x) + \mu|\Sigma_x|\log|\Sigma| \tag{5}$$

where $|\Sigma_x|$ is the number of distinct characters in x and Σ is the number of distinct characters in the input string s (and therefore also in $\hat{s} = \mathtt{bwt}(s)$). The rationale for this choice is the following: 1) we can get rid of the parameter λ since the optimal partition does not change if the cost function is scaled by a constant factor, 2) we use the term $\mu|\Sigma_x|\log|\Sigma|$ because the order zero compressor $\mathtt{A}$, in addition to the encoding of x, must somehow indicate which characters of s are present in x. We point out that this is not the only possible choice: using our library one can define a completely different cost function (see below).

User interface. Our library provides a simple interface to boost the performance of an arbitrary compressor using either mtf or the optimal partitioning strategy outlined in Sect. 3. This can be done even without any knowledge of the Burrows-Wheeler transform! The user simply needs to provide compression and decompression procedures and, for the computation of the optimal partition, a procedure evaluating the cost function $C(x)$.

Compression: the user needs to provide the procedure encode(char $*s$, int n) where s is the string to be encoded (compressed) and n is its length. Optionally, the user can provide also two procedures enc_start(void), and enc_stop(void) for initializations and cleanups. Assuming that the optimal partitioning strategy returns the partition $\hat{s}_1, \ldots, \hat{s}_k$ of $\hat{s} = \mathsf{bwt}(s)$, our library first calls the procedure enc_start() followed by encode($\hat{s}_i, |\hat{s}_i|$) for $i = 1, \ldots, k$, followed by enc_stop(). By means of a command line switch it is possible to use mtf instead of the partitioning strategy. In this case our library simply calls enc_start() followed by encode(mtf($\hat{s}$), $|\hat{s}|$), followed by enc_stop().

Decompression: the user needs to provide the procedure decode(char $*t$, int n), and, optionally, dec_start(void), and dec_stop(void). The decompression of a file consists of a call to dec_start(), followed by one or more calls to decode(), followed by a call to dec_stop(). The size of $\hat{s} = \mathsf{bwt}(s)$ is stored at the beginning of the compressed file, therefore the booster goes on calling decode() until the whole $\hat{s}$ has been recovered.

Cost function: here there are many possible choices. If the user only wants to use mtf no additional procedure is needed. As an alternative, the user can provide a procedure int cost(char $*x$, int n) that given a string x of length n returns an estimate of the size of the output produced by encode with input x. To compute the optimal partitioning according to the "real cost" model the user should define cost() so that it returns the number of bits produced by encode($x, |x|$), but nothing prevents the user to define cost in a different way. In addition, or in alternative, to cost() the user can define a procedure double bound(stats $*$) that given *the statistics* of x (i.e. the number of occurrences of each character) returns an estimate of the number of bits produced by encode($x, |x|$). To compute the optimal partitioning according to the "entropy bound" model the user should define bound() so that it returns the value $C(x)$ given by (5), but again, other alternatives are possible.

5 Experimental Results

Using the boosting library described in the previous section we have implemented several bwt-based compressors. By means of extensive experiments we tried to assess to what extent mtf and the boosting algorithm are able to turn a generic order zero compressor into a state of the art compressor. We ran all experiments on a 2.6 GHz Pentium 4 CPU with 1.5 GB of main memory running Fedora Linux. All code was written in C and compiled using gcc Ver. 3.2.2. As a testbed we used the collection of file introduced in [18] for testing suffix array construction algorithms. These files are described in Figure 4 and available for download from [17]. We used these files instead of the classical Calgary and Canterbury corpus since these corpora contain only relatively small files which provide a poor indication of the asymptotic behavior of our algorithms (the results for the largest files of the Canterbury corpus are reported in the Appendix).

Algorithms. The following is a description of the algorithms tested in our experiments.

Bzip2 is the well known tool based on the bwt developed by Julian Seward [21]. Bzip2 splits the input file into blocks of size 900Kb and computes the bwt followed by mtf on each block. The actual compression is done using rle0[2] followed by Multiple-Table Huffman coding. Note that splitting the input file into smaller blocks is a sensible design choice (for example it limits the amount of working memory used by the algorithm). However, if the input file is homogeneous it is usually advantageous, in terms of compression ratio, not to split the input. Since a single bwt is preferable also for the construction of compressed indexes, all the algorithms listed below compute the bwt of the whole file. Hence, we have included Bzip2 only for providing a reference to a known tool.

MtfRleMth executes the same steps as Bzip2 operating on the whole input instead that on fixed length blocks.

MtfRleRc. The earliest versions of Bzip2 used arithmetic coding instead of multiple-table Huffman. The use of arithmetic coding was later discontinued mainly because of possible patenting problems. Recently, range coding [19] has been (re)discovered as a patent-free alternative to arithmetic coding. Range coding and arithmetic coding are based on similar concepts and achieve similar compression. MtfRleRc compresses the bwt using mtf followed by rle0, followed by range coding (we used the implementation

[2]We use rle to denote the run length encoding of the runs of any character, while we use rle0 to denote the run length encoding only of the runs of zeros. If a string was produced by mtf, rle0 is the natural choice because of the massive presence of 0-runs as observed by Fenwick [5].

from [14]). Note that MtfRleRc is identical to MtfRleMth except that, instead of Multiple-table Huffman coding, it uses range coding.

RleRc compresses the bwt using rle followed by range coding.

BoostRleRc is an implementation of the boosting algorithm applied to the compressor consisting of rle followed by range coding. Note that the difference in compression between RleRc and BoostRleRc gives the "added value" of the use of the booster.

MtfRleAc, RleAc, BoostRleAc are analogous respectively to MtfRleRc, RleRc, BoostRleRc except that they use the arithmetic coding routines from [24] instead of range coding.

MtfRleHuff, RleHuff, BoostRleHuff are analogous respectively to MtfRleRc, RleRc, BoostRleRc except that they use Huffman coding instead of range coding. Note that MtfRleHuff differs from MtfRleMth in that the former uses a single Huffman table whereas the latter uses up to six tables for the same file.

Wavelet. This algorithm computes the bwt of the whole input and compresses the resulting string using a wavelet tree [10]. Recall that a wavelet tree is a data structure that reduces the compression of a string over a finite alphabet to the compression of a set of binary strings. The binary strings are then compressed using rle to represent runs on 0's and 1's, and the rle values are finally encoded using γ-coding (this is essentially the algorithm Wzip of [9]). The importance of wavelet trees stems from the fact that they have been used for the design of efficient bwt-based compressed indices [8, 10, 11] and that it has been recently proven in [6] that they also can achieve the k-th order entropy for any $k \geq 0$. More precisely, from [6] follows that for a string s over the alphabet Σ the output size of Wavelet is bounded by $4|s|\, H^*_k(s) + 6|\Sigma|^{k+1} \log(|s|)$ bits.

BoostWav is an implementation of the boosting algorithm applied to the wavelet tree encoder using the "real cost" model. Thus the difference between Wavelet and BoostWav is that the former builds a wavelet tree on the whole bwt, whereas the latter finds an optimal partition of the bwt and builds a wavelet tree on each substring of the optimal partition. Again, the difference in compression between Wavelet and BoostWav gives the "added value" of the use of the booster.

PPMd is an implementation of the ppm encoder by Dmitry Shkarin [23, 22] which is the current state of the art for PPM compression . In our tests we used PPMd at its maximum strength, that is using a model of order 16 and 256Mb of working memory.

PPMdr1 consists of the PPMd encoder with option -r1. Using this option shortage of memory is handled cutting off the current model instead than restarting from scratch. This strategy improves compression at the expense of running time. Again, we used an order 16 model and 256Mb of working memory.

Range/arithmetic coding variants. The behavior of range and arithmetic coding depends on two parameters: MaxFreq and Increment. The ratio between these two values essentially controls how quickly the coding "adapts" to the new statistics. For range coding we set MaxFreq = 65536 (the largest possible value) and we experimented with three different values of Increment. Setting Increment = 256 we get a range coder with FAST adaptation, with Increment = 32 we get a range coder with MEDIUM adaptation, and finally setting Increment = 4 we get a range coder with SLOW adaptation. For arithmetic coding we set MaxFreq = 16383 (the largest possible value) and Increment = 64 obtaining therefore a FAST adaptation.

Compression ratio. Figure 5 reports the compression ratio (in bits per symbol) and compression time (microseconds per symbol) for the all the algorithms mentioned above. Looking at the average compression ratio we can see that both mtf and the boosting algorithm do a good job in transforming an order zero compressor into a state-of-the-art compressor. However, our data show some unexpected behaviors. Considering the three version of range coding (with FAST, MEDIUM, and SLOW adaptation) we see that mtf achieves the best compression using MEDIUM adaptation whereas the boosting algorithm "prefers" FAST adaptation. It is also remarkable that RleRc with FAST adaptation achieves a very good compression, better indeed that mtf combined with any version of range coding (and the same is true for RleAc FAST). This means that the bwt can be compressed efficiently using rle and an order zero encoder that quickly adapts to the new statistics (and therefore quickly forgets past symbols). This is somewhat intuitive, but to our knowledge no one observed experimentally the superiority of this strategy w.r.t. mtf, and no theoretical analysis has explained or suggested such behavior. Overall the data show that the boosting algorithm is superior to mtf in terms of compression ratio, and this seems especially true with the less effective order zero compressors (for example Huffman coding). This superiority is however paid in terms of running time as discussed below.

Running time. The data in Figure 5 (top) show that for range coding the boosting algorithm with the "real cost" model is between 4 and 5 times slower than mtf in compression while there is no significant difference in decompression. For arithmetic and Huffman coding the ratio is even higher. The data in Figure 5 (bottom) show that using the "entropy bound" model the compression time decreases significantly and there is a corresponding loss in compression efficiency. Summing up, mtf and the boosting algorithm (with the two different cost models) offer three different trade offs between compression ratio and compression time: the user can choose the one most suitable for the application at hand. Figure 6 reports the resource usage of the various stages of the boosting algorithm. We can see that the most time consuming step is the optimal partition computation via the suffix tree visit both in the "real cost" and "entropy bound" models. Note also that the peak memory usage is achieved during the LCP array computation.

Wavelet tree performance. The data in Figure 5 show that the algorithms Wavelet and BoostWav roughly achieve the same compression as the algorithms based on Huffman coding (RleHuff and BoostRleHuff) and are inferior to the algorithms based on range/arithmetic encoding. It is natural to expect that Wavelet and BoostWav can reach the performance of BoostRleRc and BoostRleAc if instead of encoding the rle values using γ coding we encode them using range/arithmetic coding which are more flexible and can encode each value using a "fractional" number of bits. However, given the results in [9], the parameter settings of those order-zero encoders may play a role as they do for BoostRleRc and BoostRleAc. We will address this issue in the full paper. Finally, we point out that the similar compression ratio of Wavelet and BoostWav provide an experimental validation of the theoretical analysis of [6] which states that even using a single wavelet tree—as in the algorithm Wavelet—we already achieve the k-th order entropy.

PPMd performance. The results in Figure 5 show that PPMd outperforms all other compressors, and, for the files of the Canterbury corpus, Figure 7 shows that the Weighted Frequency Count algorithm (which is based on the bwt) compresses better than mtf, boosting, and wavelet tree algorithms. This suggests that in the field of (bwt) compression Theory is currently a step behind Practice. Although we emphasize that for the construction of compressed indexes it is essential to have simple and efficient bwt-based algorithms whose performance are theoretically guaranteed, we take these results as a stimulus for further research!

References

[1] J. Abel. Improvements to the Burrows-Wheeler compression algorithm: After BWT stages. http://citeseer.ist.psu.edu/abel03improvements.html.

[2] S. Burkhardt and J. Kärkkäinen. Fast lightweight suffix array construction and checking. In *Proc. 14th Symposium on Combinatorial Pattern Matching (CPM '03)*, pages 55–69. Springer-Verlag LNCS n. 2676, 2003.

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

[4] S. Deorowicz. Context exhumation after the BurrowsWheeler transform. *Information Processing Letters*, 95:313–320, 2005.

File	Size (Kb)	Ave. LCP	Alphabet	Description
sprot	107,048	89.08	66	Swiss prot database (original file name sprot34.dat)
rfc	113,693	93.02	120	Concatenation of RFC text files
howto	38,498	267.56	197	Concatenation of Linux Howto text files
reuters	112,022	282.07	93	Reuters news in XML format
linux	113,530	479.00	256	Tar archive containing the Linux kernel 2.4.5 source files
jdk13	68,094	678.94	113	Concatenation of html and java files from the JDK 1.3 doc.
etext99	102,809	1,108.63	146	Concatenation of Project Gutemberg etext99/*.txt files
chr22	33,743	1,979.25	5	Genome assembly of human chromosome 22
gcc	84,600	8,603.21	150	Tar archive containing the gcc 3.0 source files
w3c	101,759	42,299.75	256	Concatenation of html files from www.w3c.org

Figure 4: Files used in our experiments.

	sprot	*rfc*	*howto*	*reut*	*linux*	*jdk13*	*etext*	*chr22*	*gcc*	*w3c*	averg	ctime	dtime
Bzip2	1.660	1.496	2.069	1.193	1.480	0.563	2.206	1.954	1.267	0.788	1.424	0.53	0.14
MtfRleMth	1.376	1.207	1.762	0.797	1.326	0.366	1.896	1.856	1.094	0.539	1.167	0.96	0.46
RleRc FAST	1.345	1.146	1.712	0.762	1.296	0.362	1.787	1.825	1.070	0.545	1.129	0.90	0.48
MtfRleRc FAST	1.372	1.202	1.781	0.792	1.332	0.360	1.886	1.765	1.096	0.536	1.161	0.90	0.48
BoostRleRc FAST	1.345	1.146	1.712	0.762	1.296	0.362	1.787	1.825	1.070	0.545	1.129	4.11	0.48
RleRc MED.	1.368	1.187	1.792	0.795	1.353	0.392	1.850	1.829	1.131	0.576	1.171	0.89	0.48
MtfRleRc MED.	1.373	1.188	1.759	0.787	1.314	0.360	1.878	1.767	1.084	0.529	1.153	0.96	0.48
BoostRleRc MED.	1.360	1.167	1.759	0.775	1.337	0.372	1.812	1.828	1.107	0.566	1.152	4.13	0.49
RleRc SLOW	1.409	1.267	1.915	0.850	1.452	0.436	1.967	1.836	1.229	0.627	1.245	0.90	0.48
MtfRleRc SLOW	1.389	1.196	1.774	0.795	1.324	0.366	1.895	1.772	1.095	0.537	1.164	0.90	0.48
BoostRleRc SLOW	1.367	1.188	1.809	0.786	1.381	0.382	1.849	1.830	1.132	0.586	1.175	4.12	0.48
RleAc FAST	1.343	1.141	1.706	0.760	1.294	0.361	1.786	1.823	1.065	0.543	1.126	0.97	0.59
MtfRleAc FAST	1.371	1.196	1.775	0.790	1.330	0.359	1.884	1.763	1.089	0.532	1.158	0.94	0.53
BoostRleAc FAST	1.343	1.140	1.704	0.759	1.289	0.361	1.782	1.823	1.064	0.543	1.125	7.43	0.59
RleHuff	1.635	1.688	2.365	1.118	1.927	0.561	2.619	2.067	1.600	0.817	1.596	0.89	0.47
MtfRleHuff	1.446	1.279	1.859	0.840	1.387	0.386	2.030	1.878	1.150	0.572	1.230	0.95	0.46
BoostRleHuff	1.385	1.218	1.819	0.801	1.388	0.389	1.866	1.963	1.151	0.594	1.195	5.04	0.45
Wavelet	1.502	1.252	1.841	0.834	1.407	0.392	1.930	1.870	1.154	0.630	1.230	0.96	1.01
BoostWav	1.498	1.252	1.839	0.833	1.407	0.392	1.928	1.870	1.154	0.630	1.229	3.55	0.94
PPMd	1.301	1.157	1.660	0.794	1.145	0.380	1.732	1.749	0.964	0.465	1.080	0.60	0.66
PPMdr1	1.256	1.117	1.587	0.731	1.129	0.365	1.685	1.734	0.937	0.457	1.045	1.19	1.25

	sprot	*rfc*	*howto*	*reut*	*linux*	*jdk13*	*etext*	*chr22*	*gcc*	*w3c*	averg	ctime	dtime
BoostRleRc FAST	1.345	1.146	1.712	0.762	1.296	0.362	1.787	1.825	1.070	0.545	1.129	4.11	0.48
BoostRleRc FAST $\mu = 32$	1.355	1.149	1.715	0.766	1.300	0.371	1.788	1.831	1.075	0.557	1.134	3.04	0.48
BoostRleRc FAST $\mu = 8$	1.381	1.165	1.736	0.778	1.320	0.393	1.800	1.844	1.104	0.613	1.158	3.07	0.49
BoostRleRc MED.	1.360	1.167	1.759	0.775	1.337	0.372	1.812	1.828	1.107	0.566	1.152	4.13	0.49
BoostRleRc MED. $\mu = 32$	1.364	1.169	1.775	0.780	1.341	0.378	1.827	1.830	1.113	0.572	1.158	3.02	0.48
BoostRleRc MED. $\mu = 8$	1.375	1.172	1.762	0.781	1.345	0.392	1.815	1.836	1.118	0.617	1.166	3.02	0.48
BoostRleRc SLOW	1.367	1.188	1.809	0.786	1.381	0.382	1.849	1.830	1.132	0.586	1.175	4.12	0.48
BoostRleRc SLOW $\mu = 32$	1.378	1.204	1.851	0.799	1.401	0.394	1.883	1.830	1.163	0.602	1.194	3.02	0.48
BoostRleRc SLOW $\mu = 8$	1.385	1.205	1.832	0.801	1.406	0.416	1.863	1.831	1.163	0.671	1.203	3.05	0.48
BoostRleHuff	1.385	1.218	1.819	0.801	1.388	0.389	1.866	1.963	1.151	0.594	1.195	5.04	0.45
BoostRleHuff $\mu = 32$	1.397	1.250	1.901	0.818	1.445	0.403	1.925	1.968	1.201	0.612	1.229	2.96	0.45
BoostRleHuff $\mu = 8$	1.409	1.241	1.849	0.821	1.414	0.425	1.883	1.967	1.183	0.674	1.225	2.96	0.46

Figure 5: Experimental results for the files in Fig. 4. In each table Columns 2 to 11 report the compression for each file in bits per symbol. Column 12 reports the average compression in bits per symbol, and the last two columns report average compression and decompression time in microseconds per symbol. The table on the top reports the performance of the boosting algorithm for the "real cost" model only. The bottom table compares the "real cost" vs the "entropy bound" model: the first entry in each section reports the performance of the "real cost" model and the following entries the performance of the "entropy bound" model for $\mu = 8$ and $\mu = 16$ (the parameter μ is the one appearing in (5)).

	running time					peak memory	
	bwt	lcp	visit	cmpr	total	lcp	visit
sprot	0.70	0.60	1.67	0.11	3.08	7.01	5.00
rfc	0.60	0.51	2.35	0.11	3.57	6.86	5.00
howto	0.50	0.45	2.83	0.15	3.93	7.29	5.01
reut	1.24	0.55	1.92	0.08	3.79	6.58	5.00
linux	0.52	0.42	3.39	0.12	4.46	6.88	5.04
jdk13	1.15	0.40	2.10	0.05	3.70	6.26	5.00
etext	0.75	0.63	2.65	0.16	4.19	7.57	5.00
chr22	0.49	0.54	6.33	0.17	7.53	8.34	5.49
gcc	0.85	0.40	3.00	0.10	4.36	6.75	5.07
w3c	1.10	0.43	3.18	0.06	4.78	6.31	5.01

	running time				
	bwt	lcp	visit	cmpr	total
sprot	0.70	0.59	1.23	0.11	2.63
rfc	0.60	0.51	1.52	0.11	2.74
howto	0.50	0.46	1.86	0.15	2.96
reut	1.24	0.56	1.32	0.08	3.19
linux	0.52	0.42	2.17	0.12	3.23
jdk13	1.15	0.40	1.48	0.05	3.08
etext	0.75	0.64	1.59	0.16	3.14
chr22	0.49	0.54	0.89	0.18	2.10
gcc	0.86	0.40	1.64	0.10	3.00
w3c	1.10	0.43	2.34	0.06	3.94

Figure 6: Running time and peak memory usage for the various stages of the BoostRleRc (MEDIUM adaptation) algorithm using the "real cost" model (left) and the "entropy bound" model (right, the table only shows running times since the memory usage is the same as for the "real cost" model). The running times of the four basic steps (bwt computation, LCP array computation, optimal partition computation via suffix tree visit, actual compression using range coding) and the total running time are given in microseconds per input byte. The peak memory usage is given for the LCP array computation and the suffix tree visit which are the steps using more memory. Memory usage is reported as number of used bytes per input byte.

[5] P. Fenwick. Block sorting text compression — final report. Technical Report 130, Dept. of Computer Science, The University of Auckland New Zeland, 1996.

[6] P. Ferragina, R. Giancarlo, and G. Manzini. The myriad virtues of wavelet trees. In *Proc. of International Colloquium on Automata and Languages (ICALP)*, pages 561–572. Springer Verlag LNCS n. 4051, 2006.

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

[8] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly FM-index. In *Proc. 11th International Symposium on String Processing and Information Retrieval (SPIRE '04)*, pages 150–160. Springer-Verlag LNCS n. 3246, 2004.

[9] L. Foschini, R. Grossi, A. Gupta, and J. Vitter. Fast compression with a static model in high order entropy. In *IEEE DCC*, pages 62–71. IEEE Computer Society TCC, 2004.

[10] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symp. on Discrete Algorithms (SODA '03)*, pages 841–850, 2003.

[11] R. Grossi, A. Gupta, and J. Vitter. When indexing equals compression: Experiments on compressing suffix arrays and applications. In *Proc. 15th Annual ACM-SIAM Symp. on Discrete Algorithms (SODA '04)*, pages 636–645, 2004.

[12] H. Kaplan, S. Landau, and E. Verbin. A simpler analysis of Burrows-Wheeler based compression. In *Proc. of the 17th Symposium on Combinatorial Pattern Matching (CPM '06)*. Springer-Verlag LNCS, 2006.

[13] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. 12th Symposium on Combinatorial Pattern Matching (CPM '01)*, pages 181–192. Springer-Verlag LNCS n. 2089, 2001.

[14] M. Lundqvist. Carryless range coding. http://hem.spray.se/mikael.lundqvist/.

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

[16] G. Manzini. Two space saving tricks for linear time LCP computation. In *Proc. of 9th Scandinavian Workshop on Algorithm Theory (SWAT '04)*, pages 372–383. Springer-Verlag LNCS n. 3111, 2004.

[17] G. Manzini and P. Ferragina. Lightweight suffix sorting home page. http://www.mfn.unipmn.it/~manzini/lightweight.

[18] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40:33–50, 2004.

[19] G. N. N. Martin. Range encoding: an algorithm for removing redundancy from a digitised message. In *Video & Data Recording Conference*, 1979. Available from: http://www.compressconsult.com/rangecoder.

[20] G. Navarro and V. Mäkinen. Compressed full text indexes. Technical Report TR/DCC-2006-6, Dept. of Computer Science, University of Chile, 2006.

[21] J. Seward. The BZIP2 home page, 2006. http://www.bzip.org.

[22] D. Shkarin. PPMd compressor Ver. J. http://www.compression.ru/ds/.

[23] D. Shkarin. PPM: One step to practicality. In *IEEE Data Compression Conference*, pages 202–211, 2002.

[24] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

A Results for the Canterbury Corpus

	alice	*asyoulik*	*kennedy*	*lcet10*	*plrabn*	*ptt5*	*bible*	*ecoli*	*world*	averg	ctime	dtime
Bzip2	2.272	2.529	1.012	2.019	2.417	0.776	1.672	2.158	1.584	1.787	0.41	0.16
MtfRleMth	2.267	2.529	1.057	2.015	2.416	0.799	1.579	2.122	1.392	1.718	0.44	0.41
RleRc fast	2.328	2.572	1.500	2.052	2.418	0.730	1.539	2.126	1.393	1.740	0.46	0.50
MtfRleRc fast	2.293	2.556	0.857	2.032	2.427	0.814	1.568	2.016	1.404	1.669	0.45	0.45
BoostRleRc fast	2.328	2.572	1.500	2.052	2.418	0.730	1.536	2.126	1.391	1.739	2.93	0.45
RleRc med.	2.563	2.803	1.433	2.232	2.604	0.732	1.643	2.121	1.487	1.798	0.43	0.44
MtfRleRc med.	2.296	2.558	0.959	2.029	2.431	0.794	1.571	2.010	1.397	1.673	0.44	0.45
BoostRleRc med.	2.388	2.638	1.433	2.089	2.453	0.732	1.561	2.121	1.446	1.753	2.81	0.44
RleRc slow	2.724	2.990	1.488	2.374	2.766	0.750	1.738	2.124	1.567	1.859	0.43	0.44
MtfRleRc slow	2.316	2.587	1.055	2.050	2.457	0.794	1.586	2.010	1.406	1.689	0.44	0.45
BoostRleRc slow	2.386	2.637	1.488	2.107	2.471	0.750	1.567	2.122	1.451	1.762	2.80	0.44
RleAc fast	2.320	2.564	1.497	2.044	2.410	0.727	1.535	2.124	1.387	1.736	0.51	0.54
MtfRleAc fast	2.284	2.548	0.848	2.023	2.419	0.810	1.564	2.013	1.398	1.664	0.50	0.51
BoostRleAc fast	2.320	2.564	1.497	2.044	2.410	0.727	1.535	2.124	1.387	1.736	5.31	0.54
RleHuff	2.934	3.240	1.818	2.635	3.162	0.839	2.141	2.372	1.878	2.169	0.42	0.41
MtfRleHuff	2.377	2.670	1.500	2.132	2.570	0.834	1.679	2.114	1.453	1.801	0.42	0.41
BoostRleHuff	2.483	2.711	1.608	2.167	2.534	0.772	1.616	2.290	1.493	1.856	3.89	0.40
Wavelet	2.366	2.661	1.185	2.111	2.506	0.834	1.614	2.178	1.464	1.779	0.44	0.69
BoostWav	2.366	2.661	1.185	2.111	2.506	0.834	1.613	2.178	1.464	1.779	2.54	0.65
PPMd	2.038	2.310	1.145	1.794	2.206	0.754	1.408	2.024	1.211	1.590	0.80	0.83
PPMdr1	2.038	2.310	1.145	1.794	2.206	0.754	1.408	2.024	1.211	1.590	0.79	0.83
WFC06	2.149	2.409	0.816	1.893	2.272	0.706	1.463	1.954	1.298	—	—	—

Figure 7: Experimental results for the large files of the Canterbury corpus. The last line reports the results for the Weighted Frequency Count algorithm [J. Abel, Personal Communication] which is a recent non-trivial bwt-based algorithm.