Implicit Compression Boosting with Applications to Self-Indexing

Veli Mäkinen[1] * and Gonzalo Navarro[2] **

[1] Department of Computer Science, University of Helsinki, Finland.
vmakinen@cs.helsinki.fi
[2] Department of Computer Science, University of Chile.
gnavarro@dcc.uchile.cl

Abstract. *Compression boosting* (Ferragina & Manzini, SODA 2004) is a new technique to enhance zeroth order entropy compressors' performance to k-th order entropy. It works by constructing the Burrows-Wheeler transform of the input text, finding optimal partitioning of the transform, and then compressing each piece using an arbitrary zeroth order compressor. The optimal partitioning has the property that the achieved compression is boosted to k-th order entropy, for any k. The technique has an application to text indexing: Essentially, building a *wavelet tree* (Grossi et al., SODA 2003) for each piece in the partitioning yields a k-th order compressed full-text *self-index* providing efficient substring searches on the indexed text (Ferragina et al., SPIRE 2004). In this paper, we show that using explicit compression boosting with wavelet trees is not necessary; our new analysis reveals that the size of the wavelet tree built for the complete Burrows-Wheeler transformed text is, in essence, the sum of those built for the pieces in the optimal partitioning. Hence, the technique provides a way to do compression boosting implicitly, with a trivial linear time algorithm, but fixed to a specific zeroth order compressor (Raman et al., SODA 2002). In addition to having these consequences on compression and static full-text self-indexes, the analysis shows that a recent *dynamic* zeroth order compressed self-index (Mäkinen & Navarro, CPM 2006) occupies in fact space proportional to k-th order entropy.

1 Introduction

The indexed string matching problem is that of, given a long text $T[1, n]$ over an alphabet Σ of size σ, building a data structure called *full-text index* on it, to solve two types of queries: (a) Given a short pattern $P[1, m]$ over Σ, *count* the occurrences of P in T; (b) *locate* those *occ* positions in T. There are several classical full-text indexes requiring $O(n \log n)$ bits of space which can answer counting queries in $O(m \log \sigma)$ time (like suffix trees [1]) or $O(m + \log n)$ time (like suffix arrays [18]). Both locate each occurrence in constant time once the

* Funded by the Academy of Finland under grant 108219.
** Partially Funded by Fondecyt Grant 1-050493, Chile.

counting is done. Similar complexities are obtained with modern compressed data structures [6, 11, 9], requiring space $nH_k + o(n \log \sigma)$ bits (for some small k), where $H_k \leq \log \sigma$ is the k-th order empirical entropy of T. These indexes are often called *compressed self-indexes* refering to their space requirement and to their ability to work without the text and even fully replace it, by delivering any text substring without accessing T.

The main building blocks in compressed self-indexes are the Burrows-Wheeler transform T^{bwt} [3] and function $rank_c(T^{bwt}, i)$ that counts how many times symbol c appears in $T^{bwt}[1, i]$. Function $rank_c$ can be efficiently provided by building the *wavelet tree* [11] on T^{bwt}; this reduces the problem to *rank* queries on binary sequences, which are already studied by Jacobson [14] in his seminal work on compressed data structures. Using a more recent binary *rank* solution [23] inside wavelet trees, one almost automatically achieves a compressed self-index taking $nH_0 + o(n \log \sigma)$ bits of space [11, 9, 16]. Let us call this index *Succinct Suffix Array* (SSA) following [16].

What has remained unnoticed so far is that SSA actually takes only $nH_k + o(n \log \sigma)$ bits of space. This result makes some of the more complicated techniques to achieve the same result obsolete. However, our analysis builds on the existence of the compression-boosted version of SSA [9], as we show that the internal parts of the structures in the boosted version are compressed exactly the same way in the basic SSA. This shows a remarkable property of wavelet trees when used together with the encoding of Raman, Raman, and Rao [23].

In the following, we first define the entropy concepts more formally, then explain the encoding in [23], wavelet trees [11], Burrows-Wheeler transform [3], and compression boosting [7] in order to give our new analysis in a self-contained manner. We conclude with the application to space-efficient construction of (dynamic) full-text self-indexes.

2 Definitions

We assume our text $T = t_1 \dots t_n$ to be drawn from an alphabet $\{0, 1, \dots \sigma - 1\}$. Let n_c denote the number of occurrences of symbol c in T, i.e., $n_c = |\{i \mid t_i = c\}|$. Then the zero-order *empirical entropy* is defined as $H_0(T) = \sum_{0 \leq c < \sigma} \frac{n_c}{n} \log \frac{n}{n_c}$. This is the lower bound for the average code word length of any compressor that fixes the code words to the symbols independently of the context they appear in.

A tighter lower bound for texts is the *k-th order empirical entropy* $H_k(T)$, where the compressor can fix the code word based on the k-symbol context following the symbol to be coded.[1] Formally, it can be defined as $H_k(T) = \sum_{w \in \Sigma^k} \frac{n_w}{n} H_0(T|w)$, where n_w denotes the number of occurrences of substring

[1] It is more logical (and hence customary) to define the context as the k symbols preceding a symbol, but we use the reverse definition for technical convenience. If this is an issue, the texts can be handled reversed to obtain results on the more standard definition. It is anyway known that both definitions do not differ by much [8].

w in T and $T|w$ denotes the concatenation of the symbols appearing immediately before those n_w occurrences [19]. Substring $w = T[i+1, i+k]$ is called the k-*context* of symbol t_i. We take T here as a *cyclic string*, such that t_n precedes t_1, and thus the amount of k-contexts is exactly n.

3 Previous Results

3.1 Entropy-Bound Structures for Bit Vectors

Raman et al. [23] proposed a data structure to solve *rank* and *select* (inverse of *rank*) queries in constant time over a static bit vector $A = a_1 \ldots a_n$ with binary zero-order entropy H_0. The structure requires $nH_0 + o(n)$ bits.

The idea is to split A into *superblocks* $S_1 \ldots S_{n/s}$ of $s = \log^2 n$ bits. Each superblock S_i is in turn divided into $2 \log n$ blocks $B_i(j)$, of $b = (\log n)/2$ bits each, thus $1 \leq j \leq s/b$. Each such block $B_i(j)$ is said to belong to *class* c if it has exactly c bits set, for $0 \leq c \leq b$. For each class c, a universal table G_c of $\binom{b}{c}$ entries is precomputed. Each entry corresponds to a possible block belonging to class c, and it stores all the local *rank* answers for that block. Overall all the G_c tables add up $2^b = \sqrt{n}$ entries, and $O(\sqrt{n}\ \mathrm{polylog}(n))$ bits.

Each block $B_i(j)$ of the sequence is represented by a pair $D_i(j) = (c, o)$, where c is its class and o is the index of its corresponding entry in table G_c. A block of class c thus requires $\log(c+1) + \log \binom{b}{c}$ bits. The first term is $O(\log \log n)$, whereas all the second terms add up $nH_0 + O(n/\log n)$ bits. To see this, note that $\log \binom{b}{c_1} + \log \binom{b}{c_2} \leq \log \binom{2b}{c_1+c_2}$, and that $nH_0 \geq \log \binom{b(n/b)}{c_1+\ldots+c_{n/b}}$. The pairs $D_i(j)$ are of variable length and are all concatenated into a single sequence.

Each superblock S_i stores a pointer P_i to its first block description in the sequence (that is, the first bit of $D_i(1)$) and the *rank* value at the beginning of the superblock, $R_i = rank(A, (i-1)s)$. P and R add up $O(n/\log n)$ bits. In addition, S_i contains s/b numbers $L_i(j)$, giving the initial position of each of its blocks in the sequence, relative to the beginning of the superblock. That is, $L_i(j)$ is the position of $D_i(j)$ minus P_i. Similarly, S_i stores s/b numbers $Q_i(j)$ giving the *rank* value at the beginning of each of its blocks, relative to the beginning of the superblock. That is, $Q_i(j) = rank(A, (i-1)s + (j-1)b) - R_i$. As those relative values are $O(\log n)$, sequences L and Q require $O(n \log \log n / \log n)$ bits.

To solve $rank(A, p)$, we compute the corresponding superblock $i = 1 + \lfloor p/s \rfloor$ and block $j = 1 + \lfloor (p - (i-1)s)/b \rfloor$. Then we add the *rank* value of the corresponding superblock, R_i, the relative *rank* value of the corresponding block, $Q_i(j)$, and complete the computation by fetching the description (c, o) of the block where p belongs (from bit position $P_i + L_i(j)$) and performing a (precomputed) local *rank* query in the universal table, $rank(G_c(o), p-(i-1)s-(j-1)b)$.

The overall space requirement is $nH_0 + O(n \log \log n / \log n)$ bits, and *rank* is solved in constant time. We do not cover *select* because it is not necessary to follow this paper.

The scheme extends to sequences over small alphabets as well [9]. Let $B = a_1 \ldots a_b$ be the symbols in a block, and call n_a the number of occurences of symbol $a \in [1,q]$ in B. We call $(n_1, \ldots, n_q)$ the *class* of B. Thus, in our (c, o) pairs, c will be a number identifying the class of B and o an index within the class. A simple upper bound to the number of classes is $(b+1)^q$ (as a class is a tuple of q numbers in $[0,b]$, although they have to add up b). Thus $O(q \log\log n)$ bits suffice for c (a second bound on the number of classes is q^b as there cannot be more classes than different sequences). Just as in the binary case, the sum of the sizes of all o fields adds up $nH_0(A) + O(n/\log_q n)$ [9].

3.2 Wavelet Trees and Entropy-Bound Structures for Sequences

We now extend the result of the previous section to larger alphabets. The idea is to build a wavelet tree [11] over sequences represented using *rank* structures for small alphabets.

A binary wavelet tree is a balanced binary tree whose leaves represent the symbols in the alphabet. The root is associated with the whole sequence $A = a_1 \cdots a_n$, its left child with the subsequence of A obtained by concatenating all positions i having $a_i < \sigma/2$, and its right child with the complementary subsequence (symbols $a_i \geq \sigma/2$). This subdivision is continued recursively, until each leaf contains a repeat of one symbol. The sequence at each node is represented by a bit vector that tells which positions (those marked with 0) go to the left child, and which (marked with 1) go to the right child. It is easy to see that the bit vectors alone are enough to determine the original sequence: To recover a_i, start at the root and go left or right depending on the bit vector value B_i at the root. When going to the left child, replace $i \leftarrow rank_0(B,i)$, and similarly $i \leftarrow rank_1(B,i)$ when going right. When arriving at the leaf of character c it must hold that the original a_i is c. This requires $O(\log\sigma)$ *rank* operations over bit vectors.

It also turns out that operations *rank* and *select* on the original sequence can be carried out via $O(\log\sigma)$ operations of the same type on the bit vectors of the wavelet tree [11]. For example, to solve $rank_c(A,i)$, start at the root and go to the left child if $c < \sigma/2$ and to the right child otherwise. When going down, update i as in the previous paragraph. When arriving at the leaf of c, the current i value is the answer.

A multiary wavelet tree, of arity q, is used in [9]. In this case the sequence of each wavelet tree node ranges over alphabet $[1,q]$, and symbol rank/select queries are needed over those sequences. One needs $\log_q \sigma$ operations on those sequences to perform the corresponding operation on the original sequence.

Either for binary or general wavelet trees, it can be shown that the H_0 entropies in the representations of the sequences at each level add up to $nH_0(A)$ bits [11, 9]. However, as we have $O(\sigma)$ bit vectors, the sublinear terms sum up to $o(\sigma n)$. The space occupancy of the sublinear structures can be improved to $o(n\log\sigma)$ by concatenating all the bit vectors of the same level into a single sequence, and handling only $O(\log\sigma)$ such sequences[2]. It is straightforward to do *rank*, as well as obtaining symbol a_i, without any extra information [9].

[2] Note that $o(n\log\sigma)$ is sublinear in the size of A measured in bits.

If we now represent the concatenated bit vectors of the binary wavelet tree by the *rank* structures explained in the previous section, we obtain a structure requiring $nH_0(A) + O(n \log\log n / \log_\sigma n) = nH_0(A) + o(n \log \sigma)$ bits, solving *rank* in $O(\log \sigma)$ time. Within the same bounds one can solve *select* as well [23, 11].

Theorem 1 ([11]). *Let L be a string and B_v the corresponding binary sequence for each node v of the wavelet tree of L. Then $\sum_v |B_v| H_0(B_v) = |L| H_0(L)$.*

One can also use multiary wavelet trees and represent the sequences with alphabet size q using the techniques for small alphabets (see the end of previous section). With a suitable value for q, one obtains a structure requiring the same $nH_0(A) + o(n \log \sigma)$ bits of space, but answering *rank* and *select* in constant time when $\sigma = O(\mathrm{polylog}(n))$, and $O(\lceil \log \sigma / \log\log n \rceil)$ time in general [9].

3.3 Full-Text Self-Indexes

Many full-text self-indexes are based on representing the Burrows-Wheeler transform [3] of a text using wavelet trees to support efficient substring searches. We follow the description given in [16].

The Burrows-Wheeler Transform. The *Burrows-Wheeler transform (BWT)* [3] of a text T produces a permutation of T, denoted by T^{bwt}. We assume that T is terminated by an endmarker "\$" $\in \Sigma$, smaller than other symbols. String T^{bwt} is the result of the following transformation: (1) Form a *conceptual* matrix $\mathcal{M}$ whose rows are the cyclic shifts of the string T, call F its first column and L its last column; (2) sort the rows of $\mathcal{M}$ in lexicographic order; (3) the transformed text is $T^{bwt} = L$.

The BWT is reversible, that is, given T^{bwt} we can obtain T. Note the following properties [3]:

a. Given the i-th row of $\mathcal{M}$, its last character $L[i]$ precedes its first character $F[i]$ in the original text T, that is, $T = \ldots L[i]F[i] \ldots$.
b. Let $L[i] = c$ and let r_i be the number of occurrences of c in $L[1,i]$. Let $\mathcal{M}[j]$ be the r_i-th row of $\mathcal{M}$ starting with c. Then the character corresponding to $L[i]$ in the first column F is located at $F[j]$ (this is called the *LF mapping*: $LF(i) = j$). This is because the occurrences of character c are sorted both in F and L using the same criterion: by the text following the occurrences.

The BWT can then be reversed as follows:

1. Compute the array $C[1,\sigma]$ storing in $C[c]$ the number of occurrences of characters $\{\$, 1, \ldots, c-1\}$ in the text T. Notice that $C[c]+1$ is the position of the first occurrence of c in F (if any).
2. Define the *LF mapping* as follows: $LF(i) = C[L[i]] + rank_{L[i]}(L, i)$.
3. Reconstruct T backwards as follows: set $s = 1$ (since $\mathcal{M}[1] = \$t_1t_2\ldots t_{n-1}$) and, for each $i \in n-1, \ldots, 1$ do $T[i] \leftarrow L[s]$ and $s \leftarrow LF[s]$. Finally put the endmarker $T[n] = \$$.

The BWT transform by itself does not compress T, it just permutes its characters. However, this permutation is more compressible than the original T. Actually, it is not hard to compress L to $O(nH_k + \sigma^{k+1} \log n)$ bits, for any $k \geq 0$ [19]. The idea is as follows (we will reuse it in our new analysis later): Partition L into minimum number of pieces $L^1 L^2 \cdots L^\ell$ such that the symbols inside each piece $L^p = L[i_p, j_p]$ have the same *k-context*. Note that the k-context of a symbol $L[i]$ is $\mathcal{M}[i][1, k]$. By the definition of k-th order entropy, it follows that $|L^1|H_0(L^1) + |L^2|H_0(L^2) + \cdots + |L^\ell|H_0(L^\ell) = nH_k$. That is, if one is able to compress each piece up to its zero-order entropy, then the end result is k-th order entropy.

Theorem 2 ([19]). *Let $L = L^1 L^2 \ldots L^\ell$ be a partition of L, the BWT of T, according to contexts of length k in $\mathcal{M}$. Then $\sum_{1 \leq i \leq \ell} |L^j| H_0(L^j) = nH_k(T)$.*

Using, say, arithmetic coding on each piece, one achieves $nH_k + \sigma^{k+1} \log n$ bits encoding of T for a *fixed* k. The latter term comes from the encoding of the symbol frequencies in each piece separately. This fact is the base of *compression boosting* [7]; they give a linear time algorithm to find, for a given zero order compressor, the *optimal partitioning* of L such that when each piece is compressed using the given zero order compressor, the compression result is the best over all possible partitions. Notice that the partitions fixed by the k-contexts are a subset of all partitions, and hence the resulting compression can be bounded by k-th order entropy for *any* k.

Suffix Arrays. The *suffix array* $\mathcal{A}[1, n]$ of text T is an array of pointers to all the suffixes of T in lexicographic order. Since T is terminated by the endmarker "\$", all lexicographic comparisons are well defined. The i-th entry of $\mathcal{A}$ points to text suffix $T[\mathcal{A}[i], n] = t_{\mathcal{A}[i]} t_{\mathcal{A}[i]+1} \ldots t_n$, and it holds $T[\mathcal{A}[i], n] < T[\mathcal{A}[i+1], n]$ in lexicographic order.

Given the suffix array and T, the occurrences of the pattern $P = p_1 p_2 \ldots p_m$ can be counted in $O(m \log n)$ time. The occurrences form an interval $\mathcal{A}[sp, ep]$ such that suffixes $t_{\mathcal{A}[i]} t_{\mathcal{A}[i]+1} \ldots t_n$, for all $sp \leq i \leq ep$, contain the pattern P as a prefix. This interval can be searched for using two binary searches in time $O(m \log n)$. Once the interval is obtained, a locating query is solved simply by listing all its pointers in constant time each.

We note that the suffix array $\mathcal{A}$ is essentially the matrix $\mathcal{M}$ of the BWT (Sect. 3.3), as sorting the cyclic shifts of T is the same as sorting its suffixes given the endmarker "\$": $\mathcal{A}[i] = j$ if and only if the i-th row of $\mathcal{M}$ contains the string $t_j t_{j+1} \ldots t_{n-1} \$ t_1 \ldots t_{j-1}$.

Backward Search. The FM-index [6] is a self-index based on the Burrows-Wheeler transform. It solves counting queries by finding the interval of $\mathcal{A}$ that contains the occurrences of pattern P. The FM-index uses the array C and function $rank_c(L, i)$ of the LF mapping to perform backward search of the pattern. Fig. 1 shows the counting algorithm. Using the properties of the BWT, it is easy to see that the algorithm maintains the following invariant [6]: At the i-th phase, variables sp and ep point, respectively, to the first and last row of $\mathcal{M}$ prefixed by $P[i, m]$. The correctness of the algorithm follows from this observation. Note that P is processed backwards, from p_m to p_1.

```
Algorithm FMCount($P[1, m]$,$L[1, n]$)
(1)  $i \leftarrow m$;
(2)  $sp \leftarrow 1$; $ep \leftarrow n$;
(3)  while ($sp \leq ep$) and ($i \geq 1$) do
(4)        $c \leftarrow P[i]$;
(5)        $sp \leftarrow C[c] + rank_c(L, sp - 1)+1$;
(6)        $ep \leftarrow C[c] + rank_c(L, ep)$;
(7)        $i \leftarrow i - 1$;
(8)  if ($ep < sp$) then return 0 else return $ep - sp + 1$;
```

Fig. 1. FM-index algorithm for counting the number of occurrences of $P[1, m]$ in $T[1, n]$.

Note that array C can be explicitly stored in little space, and to implement $rank_c(L, i)$ in little space we can directly use the wavelet tree as explained in Sect. 3.2. The space usage is $nH_0 + o(n \log \sigma)$ bits and the m steps of backward search take overall $O(m \log \sigma)$ time [16].

4 Implicit Compression Boosting

In Sect. 3.3 we showed that if L is partitioned into ℓ pieces $L^1 L^2 \cdots L^\ell$ according to the k-contexts, then it is enough to achieve zero-order entropy within each partition to obtain k-th order total entropy. We now prove that the simple solution of Sect. 3.3 supporting backward search requires only nH_k bits of space. We start with an important lemma.

Lemma 1. *Let $L = L^1 L^2 \cdots L^\ell$ be any partition of L, the BWT of T. The number of bits used by a partition L^j in the wavelet tree of L is upper bounded by $|L^j| H_0(L^j) + O(|L^j| \log \sigma \log \log n / \log n + \sigma \log n)$.*

Proof. The bits corresponding to L^j form a substring of the bit vectors at each node of the wavelet tree, as their positions are mapped to the left and right child using $rank_0$ or $rank_1$, thus order is preserved. Let us consider a particular node of the wavelet tree and call B its bit sequence. Let us also call B^j the substring of B corresponding to partition L^j, and assume B^j has l^j bits set. Consider the blocks of b bits that compose B, according to the partitioning of [23] (Section 3.1). Let $B^j_{blk} = B^j_1 B^j_2 \ldots B^j_t$ be the concatenation of those bit blocks that are *fully contained* in B^j, so that B^j_{blk} is a substring of B^j of length $b \cdot t$. Assume B^j_i has l^j_i bits set, so that B^j_{blk} has $l^j_1 + \ldots + l^j_t \leq l^j$ bits set. The space the o fields of the (c,o) representations of blocks B^i_j take in the compressed B^j_{blk} is

$$\sum_{i=1}^{t} \left\lceil \log \binom{b}{l^j_i} \right\rceil \;\leq\; \log \binom{b \cdot t}{l^j_1 + \ldots + l^j_t} + t \;\leq\; \log \binom{|B^j|}{l^j} + t \;\leq\; |B^j| H_0(B^j) + t$$

where all the inequalities hold by simple combinatorial arguments [21] and have been reviewed in Section 3.1.

Note that those B^j bit vectors are precisely those that would result if we built the wavelet tree just for L^j. According to Theorem 1, adding up those $|B^j|H_0(B^j)$ over all the $O(\sigma)$ wavelet tree nodes gives $|L^j|H_0(L^j)$. To this we must add three space overheads. The first is the extra t bits above, which add up $O(|L^j| \log \sigma / \log n)$ over the whole wavelet tree because $b \cdot t \leq |B^j|$ and the $|B^j|$ lengths add up $|L^j|$ at each wavelet tree level. The second overhead is the space of the blocks that overlap with B^j and thus were not counted: As B^j is a substring of B, there can be at most 2 such blocks per wavelet tree node. At worst they can take $O(\log n)$ bits each, adding up $O(\sigma \log n)$ bits over the whole wavelet tree. The third overhead is that of the c fields, which add up $O(|L^j| \log \sigma \log \log n / \log n)$.

The above lemma lets us split the wavelet tree "horizontally" into pieces. Let us add up all the zero-order entropies for the pieces. If we partition L according to contexts of length k in $\mathcal{M}$, and add up all the space due to all partitions in the wavelet tree, we get $\sum_{1 \leq j \leq \ell} |L^j|H_0(L^j) = nH_k(T)$ (Theorem 2). To this we must add *(i)* $O(|L^j| \log \sigma / \log n)$, which sums up to $O(n \log \sigma / \log n) = o(n \log \sigma)$ bits over all the partitions; *(ii)* $O(\sigma \log n)$ bits per partition, which gives $O(\ell \sigma \log n)$; and *(iii)* $O(|L^j| \log \sigma \log \log n / \log n)$, which sums up to $O(n \log \sigma \log \log n / \log n)$ $= o(n \log \sigma)$. In the partitioning we have chosen we have $\ell \leq \sigma^k$, thus the upper bound $nH_k + o(n \log \sigma) + O(\sigma^{k+1} \log n)$ holds for the total number of bits spent in the wavelet tree. The next theorem immediately follows.

Theorem 3. *The space required by the wavelet tree of L, the BWT of T, if the bitmaps are compressed using [23], is $nH_k(T) + o(n \log \sigma) + O(\sigma^{k+1} \log n)$ bits for any $k \geq 0$. This is $nH_k(T) + o(n \log \sigma)$ bits for any $k \leq \alpha \log_\sigma n - 1$ and any constant $0 < \alpha < 1$. Here n is the length of T and σ its alphabet size.*

Note that this holds *automatically and simultaneously for any* k, and we do not even have to care about k in the index. Fig. 2 illustrates. Our result obviously applies as well to the BWT alone, without wavelet tree on top, if we use a suitable local zero-order encoder [9].

5 Discussion

We have shown that the space produced by *any* splitting of L into pieces is achieved in the simple arrangement having just one wavelet tree. In [7] they

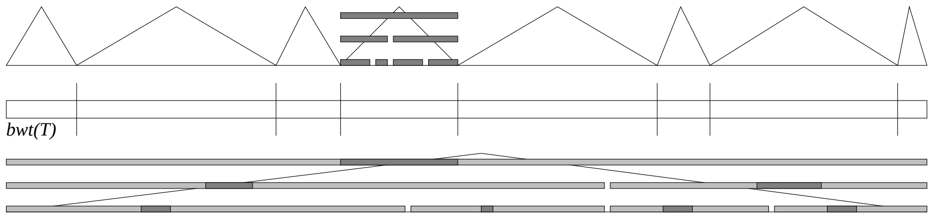

Fig. 2. Illustration of the argument used in the theorem. On top of $bwt(T) = T^{bwt}$, the wavelet trees induced by the optimal partitioning. One of those is highlighted so that we show the bitmaps stored in the wavelet tree. Below $bwt(T)$, a single wavelet tree built for the whole sequence. The bitmaps of this large wavelet tree are also shown, and they contain the bitmaps of the small highlighted wavelet tree on top.

introduce an algorithm to find the optimal partitioning. We have just used their analysis to show that it is not necessary to apply such a partitioning algorithm to achieve the boosted result. Their technique, on the other hand, has more general applications unrelated to wavelet trees.

Several full-text self-indexes in the literature build on the wavelet tree of the BWT of the text [16, 9], and engage in different additional arrangements to reach k-th order compression. In [16], they first run-length compress the BWT in order to reduce its length to $O(nH_k)$ and then apply the BWT. In [9] they explicitly cut the BWT into pieces L^j so that the sum of nH_0 sizes of the pieces adds up nH_k. In both cases, the simpler version they build on (just the wavelet tree of the BWT) would have been sufficient. Thus, we have achieved a significant simplification in the design of full-text indexes.

Also the paper where the wavelet tree is originally proposed [11] as an internal tool to design one of the most space-efficient compressed full-text indexes, would benefit from our simplification. They cut L into a table of *lists* (columns) and *contexts* (rows). All the entries across a row correspond to a contiguous piece of L, that is, some context L^j. A wavelet tree is built over each table row so as to ensure, again, that the sum of zero-order entropies over the rows adds up to global k-th order entropy. Our finding implies that all rows could have been concatenated into a single wavelet tree and the same space would have been achieved. This would greatly simplify the original arrangement. Interestingly, in [12] they find out that, if they use gap encoding over the successive values along a *column*, and they then concatenate all the columns, the total space is $O(nH_k)$ without any table partitioning as well. Both results stem from the same fact: the cell entropies can be added in any order to get nH_k.

Finally, it is interesting to point out that, in a recent paper [5], the possibility of achieving k-th order compression when applying wavelet trees over the BWT is explored (among many other results), yet they resort to run-length compression to achieve this. Once more, our finding is that this is not really necessary to achieve k-th order compression if the levels of the wavelet tree are represented using the technique of block identifier encoding [22].

6 Application to Space-Efficient Construction of (Dynamic) Self-Indexes

Another consequence of our result is that we obtain an $O(n \log n \log \sigma)$ time construction algorithm for a compressed self-index requiring $nH_k + o(n \log \sigma)$ bits *working space* during construction: This is obtained by enhancing our recent result on dynamic compressed self-indexes:

Theorem 4 ([17]). *There is a data structure maintaining a collection $\{T_1, T_2, \dots T_m\}$ of texts in $nH_0(\mathcal{C}) + o(n \log \sigma)$ bits supporting counting of occurrences of a pattern P in $O(|P| \log n \log \sigma)$ time, and inserting and deleting a text T in $O(|T| \log n \log \sigma)$ time. After counting, any occurrence can be located in time $O(\log^2 n \log \log n)$. Any substring of length ℓ from any T in the collection can be displayed in time $O(\log n(\ell \log \sigma + \log n \log \log n))$. Here n is the length of the concatenation $\mathcal{C} = 0\ T_1 0\ T_2 \cdots\ 0\ T_m$, and we assume $\sigma = o(n)$.*

The dynamic index above uses wavelet tree with the static encoding [23] (see Sect. 3.1) replaced with a dynamic version of the same encoding: The dynamic bit vector representation [17] achieves the same $nH_0 + o(n)$ space as the static, but supports *rank* and *select*, and in addition insertions and deletions of bits, in $O(\log n)$ time. This can then be used to improve the dynamic index of Chan et al. [4] to obtain the above result.

Exactly the same analysis as in Sect. 4 applies to this dynamic variant, and Theorem 4 is boosted into the following.

Corollary 1. *There is a data structure maintaining a collection $\{T_1, T_2, \dots T_m\}$ of texts in $nH_k(\mathcal{C}) + o(n \log \sigma)$ bits, for any $k \leq \alpha \log_\sigma n - 1$ and constant $0 < \alpha < 1$, supporting the same operations of Theorem 4 within the same complexities.*

Now, just inserting text T into an empty collection, yields the promised space-efficient construction algorithm for compressed self-index. This index can be easily converted into a more efficient static self-index, where a static wavelet tree requires the same space and reduces the $O(\log n \log \sigma)$ time complexities to $O(\lceil \log \sigma / \log \log n \rceil)$ [9].

Therefore, we have obtained the *first* compressed self-index with space essentially equal to the k-th order empirical entropy of the text collection, which in addition can be built within this working space. Alternative dynamic indexes or constructions of self-indexes [6, 13, 2] achieve at best $O(nH_k)$ bits of space (with constants larger than 4), and in many cases worse time complexities.

Note also that, from the dynamic index just built, it is very easy to obtain the BWT of T. It is a matter of finding the characters of L one by one. This takes $O(n \log n \log \sigma)$ time, just as the construction, and gives an algorithm to build the BWT of a text within entropy bounds. The best result in terms of space complexity takes $O(n)$ bits working space, $O(n \log^2 n)$ time in the worst case, and $O(n \log n)$ time in the expected case [15]. Using $O(n \log \sigma)$ working space, there is a faster algorithm achieving $O(n \log \log \sigma)$ time requirement [13]. Finally, one can achieve the optimal $O(n)$ time with the price of $O(n \log^\epsilon n \log \sigma)$ bits of space, for some $0 < \epsilon < 1$ [20].

7 Final Practical Considerations

Our main finding is that all the sophistications [16, 9, 11] built over the simple "wavelet tree on top of the BWT" scheme in order to boost its zero-order to high-order compression are unnecessary; the basic arrangement already achieves high-order entropy if combined with a local zero-order encoder [23].

Still, the sophisticated techniques have practical value. In the actual implementation of such methods (*Pizza&Chili* site, *http://pizzachili.dcc.uchile.cl*), zero-order entropy is achieved by using *uncompressed* bit streams over a *Huffman-tree shaped* wavelet tree, instead of *compressed* bit streams over a *balanced* wavelet tree. In this case the locality property does not hold, and high-order entropy would not be achieved if just the simple wavelet tree of the BWT was used.

Huffman-shaped trees were chosen to reach zero-order compression because of the considerable difficulty in implementing Raman et al.'s scheme [23]. As both alternatives were believed to yield similar compression ratios, the Huffman-shaped option seemed to be far more attractive from a practical point of view.

The situation is rather different now that we know that Raman et al.'s scheme yields high-order by itself, thus avoiding the need of any further complication to achieve high-order compression such as run-length compression (Run-Length FM-index, RLFM [16]) or compression boosting plus managing multiple wavelet trees (Alphabet-Friendly FM-index, AFFM [9]). Those complications not only make the implementation effort comparable to that of using Raman et al.'s scheme, but also involve a considerable space overhead for extra structures.

A prototype combining Raman et al.'s compression with balanced wavelet trees has already been implemented by Francisco Claude, a student of the second author. Unlike the existing implementations, this one offers a space-time tradeoff, related to the partial sums sampling rate. A preliminary comparison with the implementations of the SSA (bare Huffman-shaped wavelet tree over the BWT), RLFM, and AFFM, shows that our technique is 2–3 times slower for counting when using the same amount of space, which confirms the original predictions about implementation overheads. In exchange, it can still operate with reasonable efficiency using less than 75% of the space needed by the alternatives. This makes it a relevant choice when space reduction is the main concern.

It is interesting that our technique can achieve such a low space even when it has to pay for space overheads like the c components in the (c, o) pairs. This opens the door to the study of other alternatives that, even when they do not surpass the "$nH_k + o(n \log \sigma)$ for $k \leq \alpha \log_\sigma n$" theoretical barrier, do behave better in practice. We point out that this barrier is not as good as it may seem when one puts numbers to the condition on k and realizes that the achievable k values are rather low. Worse than that, it is unlikely that this theoretical limit can be sensibly improved [10]. Yet, those limits are worst-case, and different methods may not have to pay the $\Theta(\sigma^{k+1} \log n)$ space overhead in practice. For example, in our case, this overhead comes from the fact that we are unable to analyze better the compression of blocks that are split among contexts, and thus we assume the worst about them. On the other hand, the c components are real space overhead in our scheme ($2n$ bits!), and that perhaps could be improved.

References

1. A. Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO ISI Series, pages 85–96. Springer-Verlag, 1985.
2. D. Arroyuelo and G. Navarro. Space-efficient construction of LZ-index. In *Proc. ISAAC'05*, pages 1143–1152, 2005.
3. M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report Technical Report 124, Digital Equipment Corporation, 1994.
4. H.-L. Chan, W.-K. Hon, and T.-W. Lam. Compressed index for a dynamic collection of texts. In *Proc. CPM'04*, pages 445–456, 2004.
5. P. Ferragina, R. Giancarlo, and G. Manzini. The myriad virtues of wavelet trees. In *Proc. ICALP'06*, pages 560–571, 2006.
6. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. FOCS'00*, pages 390–398, 2000.
7. P. Ferragina and G. Manzini. Compression boosting in optimal linear time using the Burrows-Wheeler transform. In *Proc. SODA'04*, pages 655–663, 2004.
8. P. Ferragina and G. Manzini. Indexing compressed texts. *J. of the ACM*, 52(4):552–581, 2005.
9. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM TALG*, 3(2):article 20, 2007.
10. T. Gagie. Large alphabets and incompressibility. *IPL*, 99(6):246–251, 2006.
11. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA'03*, pages 841–850, 2003.
12. R. Grossi, A. Gupta, and J. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. In *Proc. SODA'04*, pages 636–645, 2004.
13. W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a time-and-space barrier in constructing full-text indexes. In *Proc. FOCS'03*, pages 251–260, 2003.
14. G. Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS'89*, pages 549–554, 1989.
15. J. Kärkkäinen. Fast BWT in small space by blockwise suffix sorting. In *Proc. DIMACS Working Group on the Burrows-Wheeler Transform*, 2004.
16. V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic J. of Computing*, 12(1):40–66, 2005.
17. V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. In *Proc. CPM'06*, pages 307–318, 2006.
18. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. on Computing*, pages 935–948, 1993.
19. G. Manzini. An analysis of the Burrows-Wheeler transform. *J. of the ACM*, 48(3):407–430, 2001.
20. J. C. Na. Linear-time construction of compressed suffix arrays using $o(n \log n)$-bit working space for large alphabets. In *Proc. CPM'05*, pages 57–67, 2005.
21. R. Pagh. Low redundancy in dictionaries with $O(1)$ worst case lookup time. In *Proc. ICALP'99*, pages 595–604, 1999.
22. R. Raman, V. Raman, and S. Srinivasa Rao. Succinct dynamic data structures. In *Proc. WADS'01*, pages 426–437, 2001.
23. R. Raman, V. Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. SODA'02*, pages 233–242, 2002.