Compressed Representations of Sequences and Full-Text Indexes*

Paolo Ferragina[1], Giovanni Manzini[2], Veli Mäkinen[3], and Gonzalo Navarro[4]

[1] Dipartimento di Informatica, Università di Pisa, Italy.
[2] Dipartimento di Informatica, Università del Piemonte Orientale, Italy.
[3] Department of Computer Science, University of Helsinki, Finland.
[4] Center for Web Research, Department of Computer Science, University of Chile, Chile.

Abstract. Given a sequence $S = s_1 s_2 \dots s_n$ of integers smaller than $r = O(\text{polylog}(n))$, we show how S can be represented using $nH_0(S) + o(n)$ bits, so that we can know any s_q, as well as answer *rank* and *select* queries on S, in constant time. $H_0(S)$ is the zero-order empirical entropy of S and $nH_0(S)$ provides an Information Theoretic lower bound to the bit storage of any sequence S via a fixed encoding of its symbols. This extends previous results on binary sequences, and improves previous results on general sequences where those queries are answered in $O(\log r)$ time. For larger r, we can still represent S in $nH_0(S) + o(n \log r)$ bits and answer queries in $O(\log r / \log \log n)$ time.
Another contribution of this paper is to show how to combine our compressed representation of integer sequences with an existing compression boosting technique to design *compressed full-text indexes* that scale well with the size of the input alphabet Σ. Namely, we design a variant of the FM-index that indexes a string $T[1,n]$ within $nH_k(T) + o(n)$ bits of storage, where $H_k(T)$ is the k-th order empirical entropy of T. This space bound holds simultaneously for all $k \le \alpha \log_{|\Sigma|} n$, constant $0 < \alpha < 1$, and $|\Sigma| = O(\text{polylog}(n))$. This index counts the occurrences of an arbitrary pattern $P[1,p]$ as a substring of T in $O(p)$ time; it locates each pattern occurrence in $O(\log^{1+\varepsilon} n)$ time, for any constant $0 < \varepsilon < 1$; and it reports a text substring of length ℓ in $O(\ell + \log^{1+\varepsilon} n)$ time.
Compared to all previous works, our index is the first one that removes the alphabet-size dependance from all query times, in particular counting time is linear in the pattern length. Still, our index uses essentially the same space of the k-th order entropy of the text T, which is the best space obtained in previous work. We can also handle larger alphabets of size $|\Sigma| = O(n^\beta)$, for any $0 < \beta < 1$, by paying $o(n \log |\Sigma|)$ extra space and by multiplying all query times by $O(\log |\Sigma| / \log \log n)$.

1 Introduction

Recent years have witnessed an increasing interest on *succinct* data structures. Their aim is to represent the data using as little space as possible, yet efficiently answering queries on the represented data. Several results exist on the representation of sequences [24, 32, 3, 36, 37], trees [33, 12, 7], graphs [33], permutations [34], and texts [20, 8, 38, 35, 18], to name a few.

One of the most basic structures, which lie at the heart of the representation of more complex ones, are binary sequences, with *rank* and *select* queries. Given a binary sequence $S = s_1 s_2 \dots s_n$, we denote by $\mathsf{Rank}_c(S, q)$ the number of times the bit c appears in $S[1, q] = s_1 s_2 \dots s_q$, and by $\mathsf{Select}_c(S, q)$ the position in S of the q-th occurrence of bit c. The best current results [36, 37] answer those queries in constant time, retrieve any s_q in constant time, and occupy $nH_0(S) + o(n)$ bits of storage, where $H_0(S)$ is the zero-order empirical entropy of S. This space bound includes that for representing S itself, so the binary sequence is being represented in compressed form yet allowing those queries to be answered optimally.

* A subset of the results presented in this paper appeared in the *Proceedings of the Symposium on String Processing and Information Retrieval (SPIRE)*, pages 150–160, 2004. Partially supported by MIUR Project "Algorithms for the Next Generation Internet and Web (ALGO-NEXT)", by the Academy of Finland under grant 108219 (part of the work was done while the third author was visiting Bielefeld University), and by the Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile.

For the general case of sequences over an arbitrary alphabet of size r, the only known result is the one in [18] which still achieves $nH_0(S) + o(n)$ space occupancy. The data structure in [18] is the elegant *wavelet tree*, it takes $O(\log r)$ time to answer $\mathsf{Rank}_c(S, q)$ and $\mathsf{Select}_c(S, q)$ queries, and to retrieve any character s_q.

Our first contribution is a new compressed representation for general sequences, which uses $nH_0(S) + o(n)$ bits of space and answers the above queries in constant time. This generalizes previous results on binary sequences [37] and improves the existing result on general sequences [18]. Our result holds when the alphabet size is polylogarithmic in the sequence length, that is, $r = O(\text{polylog}(n))$. For larger values of r, we can represent S using $nH_0(S) + o(n \log r)$ bits of space and answer all previous queries in $O(\log r / \log\log n)$ time.

General sequences can be regarded, of course, as texts over an alphabet Σ. The difference lies on the types of queries that are of interest on texts. A *full-text index* is a data structure built over a text string $T[1, n]$ that supports the efficient search for arbitrary patterns as *substrings* of the indexed text. A full-text index is called *compressed* if its space occupancy is bounded by $\lambda n H_k(T) + o(n)$ bits for some $k \geq 0$, where λ is a constant and $H_k(T)$ is the k-th order entropy of T (see Section 4.1). If such an index also encapsulates the text, without requiring its explicit storage, then it is called a *compressed self-index*. Note that a self-index must, in addition to the search functionality, permit the display of any text substring, as the text is not separately available.

Recently, there has been a good deal of activity around compressed full-text indexes, because of their obvious applications in text databases. The most succinct self-index up to date [18] occupies $nH_k(T) + O(n \log\log n / \log_{|\Sigma|} n)$ bits, *for a fixed* $k \leq \alpha \log_{|\Sigma|} n$ and constant $0 < \alpha < 1$. It can count the number of occurrences of a pattern $P[1, p]$ in $O(p \log |\Sigma| + \text{polylog}(n))$ time, and can locate each such occurrence in $O(\log |\Sigma| \log^2 n / \log\log n)$ time. To display a text substring of length ℓ it takes the locate time plus $O(\ell \log |\Sigma|)$.

Our second contribution (which builds on the first) is a new compressed self-index that uses $nH_k(T) + o(n)$ bits *for any* $k \leq \alpha \log_{|\Sigma|} n$ and for alphabets of size $|\Sigma| = O(\text{polylog}(n))$. Our index improves over the one in [18] by removing from the query times the dependence on the alphabet size and the polylogarithmic terms. More precisely: counting takes $O(p)$ time, locating an occurrence takes $O(\log^{1+\varepsilon} n)$ time for any constant $\varepsilon > 0$, displaying a length-ℓ text substring takes the time for locating plus $O(\ell)$. For alphabets larger than $O(\text{polylog}(n))$ (and up to $O(n^{\beta})$ for $0 < \beta < 1$), our space requirement becomes $nH_k(T) + o(n \log |\Sigma|)$ bits and our query times grow by a multiplicative factor $O(\log |\Sigma| / \log\log n)$.

The rest of the paper is organized as follows. Section 2 explains our contributions in more detail and relates them to the current literature. Section 3 presents our new representation for general sequences, while Section 4 deploys such a sequence representation to design our new compressed self-index. Finally, Section 5 discusses conclusions and future directions of interesting research.

2 Our Contribution in Context

2.1 Succinct Sequence Representations

The first results on binary sequences [24, 32, 3] achieved constant time on *rank* and *select* queries by using $n + o(n)$ bits. In those schemes, n bits are used by S itself and $o(n)$ additional bits are needed by the auxiliary data structures that support the Rank_c and Select_c queries. Further refinements [36, 37] achieved constant time on the same queries by using a compressed representation of S

requiring $nH_0(S) + o(n)$ bits overall (i.e. for S and the auxiliary data structures). These solutions also support the constant-time retrieval of any bit s_q given q.

The case of general sequences, whose symbols are drawn from the range $[1, r]$, has received less attention. The only existing proposal is the *wavelet tree* [18], a powerful and elegant data structure that allows reducing *rank* and *select* operations over general sequences to *rank* and *select* operations over binary (compressed) sequences. By using the results in [36, 37] for binary sequence representations, the wavelet tree represents S in $nH_0(S)+o(n \log r)$ bits and answers s_q, $\mathsf{Rank}_c(S, q)$ and $\mathsf{Select}_c(S, q)$ queries in $O(\log r)$ time.

In this paper we generalize the result on binary sequences [36, 37] to sequences of integers in the range $[1, r]$, and obtain an improved result. The main challenge in this generalization is to generate short descriptions for pieces of the sequence, which can be computed in constant time and used to index into tables containing partial precomputed queries. This is significantly more complex than for binary sequences. We obtain a compressed sequence representation using $nH_0(S)+ O((rn \log \log n)/ \log_r n)$ bits which answers queries s_q, $\mathsf{Rank}_c(S, q)$ and $\mathsf{Select}_c(S, q)$ in constant time.

This first result is interesting only for small $r = o(\log n/ \log \log n)$, as otherwise the term $O((rn \log \log n)/ \log_r n)$ of the space complexity is $\Omega(n \log r)$. We then use that sequence representation as a basic block within a *generalized wavelet tree* to obtain better results. In fact, the original wavelet tree [18] is a *binary* tree whose nodes contain binary sequences representing the original (general) sequence restricted to a subset of its symbols. In this paper we consider a *h-ary generalization* of the wavelet tree that, in turn, requires the efficient storage into its nodes of integer sequences from the range $[1, h]$. By setting $h = O(\log^{\delta} n)$ with $\delta < 1$, and using our previous sequence representation to store the wavelet-tree nodes, we obtain a representation for S that uses $nH_0(S) + o(n)$ bits and (optimal) constant query time for any $r = O(\text{polylog}(n))$. That is, the queries answered in logarithmic time with binary wavelet trees [18] are now answered in constant time, within the same asymptotic space occupancy. For larger alphabets, we can still obtain $o(n \log r)$ extra space over the $nH_0(S)$ term and $O(\log r/ \log \log n)$ query time, again improving upon binary wavelet trees of a sublogarithmic factor.

Note that there are even better results for binary sequences [37], which we have not generalized. For example, it is possible to represent a sequence with m bits set taking $nH_0(S)+o(m)+O(\log \log n)$ bits of space, and constant query time. This is better than the result we have focused on when m is much smaller than n.

2.2 Compressed Full-Text Indexes

The classical full-text indexes, namely *suffix trees* and *suffix arrays* [4, 30, 15], are not succinct nor self-indexes: They both occupy $\Theta(n \log n)$ bits, plus $O(n \log |\Sigma|)$ bits to represent the indexed text.

The FM-index [8] has been the first self-index in the literature to achieve a space occupancy proportional to the k-th order entropy of $T[1, n]$. Precisely, the FM-index occupies at most $5nH_k(T) + o(n)$ bits of storage, and allows one to count the number of occurrences of a pattern $P[1, p]$ within T in $O(p)$ time. Each such occurrence can be located in $O(\log^{1+\varepsilon} n)$ time, where $\varepsilon > 0$ is an arbitrary constant chosen when the index is built. The FM-index can display any text substring of length ℓ in $O(\ell+\log^{1+\varepsilon} n)$ time. The design of the FM-index is based upon the relationship between the Burrows-Wheeler compression algorithm [1] and the suffix array data structure [30, 15]. It is therefore a sort of *compressed suffix array* that takes advantage of the compressibility of the indexed text in order to achieve space occupancy related to the Information Theoretic minimum. Indeed, the design of the FM-index does not depend on the parameter k whereas its space bound holds *simultaneously* for any fixed k. These remarkable theoretical properties have been validated by experimental results [9, 22] and applications [21, 41].

The above time and space bounds for the FM-index have been obtained by assuming that the size of the input alphabet is a *constant*. Hidden in the big-O notation there is an exponential dependence on the alphabet size in the space bound, and a linear dependence on the alphabet size in the time bounds. More specifically, the time to locate an occurrence is $O(|\Sigma| \log^{1+\varepsilon} n)$ and the time to display a text substring is $O((\ell + \log^{1+\varepsilon} n)\ |\Sigma|)$. In practical implementations of the FM-index [9] these dependencies have been removed in exchange for a penalty factor (usually $O(\log n)$) that multiplies all query times, including that for counting.

We point out that the FM-index *concept* is more general than the *implementation* associated to its initial proposals [8, 9]. For the sake of presentation, let us denote now by T^{bwt} the permutation of the text T given by the Burrows-Wheeler transform (see [1] and Section 4.2). Assume we implement the computation of $\mathsf{Rank}_c(T^{bwt}, q)$ and the retrieval of $T^{bwt}[q]$ in time t_{rnk} and t_{ret}, respectively, using $O(N)$ space (where the parameter N is discussed below). The general FM-index concept gives us immediately a succinct self-index that requires $O(N + n/\log^{\varepsilon} n)$ space, counts the pattern occurrences in $O(p\, t_{rnk})$ time, locates any such occurrence in $O((t_{rnk} + t_{ret}) \log^{1+\varepsilon} n)$, and displays any text substring of length ℓ in $O((t_{rnk}+t_{ret})(\ell+\log^{1+\varepsilon} n))$ time (here and in the rest of this section ε is a positive parameter chosen when the index is built). Up to date, several implementations of the FM-index concept exist:

1. The original one [8], with $t_{rnk} = O(1)$ and $t_{ret} = O(|\Sigma|)$, where $N = 5nH_k(T) + o(n)$ contains an exponential dependence on $|\Sigma|$ in the $o(n)$ term.
2. The one obtained by using the binary wavelet tree [18] to represent T^{bwt} and to implement the required functions. This yields $t_{rnk} = t_{ret} = O(\log |\Sigma|)$, so counting time deteriorates but locating and displaying times improve. The space occupancy becomes $N = nH_0(T) + o(n \log |\Sigma|)$, which depends only mildly on $|\Sigma|$ but has a main term significantly worse than in the original implementation (since here $H_0(T)$ occurs in place of $H_k(T)$). Some variants providing $t_{rnk} = t_{ret} = O(H_0(T))$ on average, and $O(\log |\Sigma|)$ in the worst case, have also been proposed [27].
3. A variant of the Compressed Suffix Array (CSA) introduced by Sadakane [39]. Although the CSA was originally conceived as completely different from the FM-index, this specific variant is actually an implementation of the FM-index concept. It represents T^{bwt} via $|\Sigma|$ binary sequences T_c^{bwt}, each indicating the occurrences of a different character in T^{bwt}. Those sequences are represented with existing techniques [36], so that $\mathsf{Rank}_c(T^{bwt}, q) = \mathsf{Rank}_1(T_c^{bwt}, q)$ can be answered in constant time. This yields $t_{rnk} = O(1)$ and $t_{ret} = O(|\Sigma|)$ as in the original implementation. The space occupancy is $N = nH_0(T) + O(n)$. The scheme works for $|\Sigma| = O(\text{polylog}(n))$.
4. The one obtained by Huffman-compressing T^{bwt} and then representing the resulting binary sequence with the techniques in [32]. This yields $t_{rnk} = t_{ret} = O(H_0(T))$ on average and $O(\log |\Sigma|)$ in the worst case. The space occupancy is $N = 2nH_0(T) + O(n)$ [17].
5. The one obtained by compressing via run-length encoding the sequence T^{bwt} [27, 28]. This approach obtains either $t_{rnk} = O(1)$ and $t_{ret} = O(|\Sigma|)$ if $|\Sigma| = O(\text{polylog}(n))$, or $t_{rnk} = t_{ret} = O(\log |\Sigma|)$ if $|\Sigma| = o(n/\log n)$. The space occupancy is $N = n(H_k(T) \log |\Sigma| + 1 + o(1))$ bits. Notice that this is the only alternative to the original FM-index with space occupancy related to the k-th order entropy (although it shows a linear dependence on n but a milder dependence on $|\Sigma|$).

We note that, by directly applying our new sequence representation on T^{bwt}, we immediately obtain $t_{rnk} = t_{ret} = O(1)$ time and $N = nH_0(T) + o(n)$, which supersedes all the alternatives whose space requirement is proportional to the zero-order entropy of T (that is, items 2, 3, and 4 above), as long as $|\Sigma| = O(\mathrm{polylog}(n))$. Yet, in this paper we do better than this by making a further step that turns $H_0(T)$ into $H_k(T)$ in the previous space bound. Precisely, our main result is an implementation of the FM-index concept with $t_{rnk} = t_{ret} = O(1)$ time and $N = nH_k(T) + o(n)$, for a reasonable range of values of k (see below). Except for the limitation $|\Sigma| = O(\mathrm{polylog}(n))$, this implementation supersedes *all* the existing implementations of the FM-index concept, so it can be regarded as *the ultimate* implementation of the FM-index. To obtain this result, we combine our sequence representation with the *compression boosting* technique introduced in [6, 10]. Compression boosting partitions the Burrows-Wheeler transformed text into contiguous areas in order to maximize the overall compression achievable with zero-order compressors used over each area. Then we use our new sequence representation in each area. The resulting structure is thus simple. It indexes a string $T[1, n]$ drawn from an alphabet Σ, with $|\Sigma| = O(\mathrm{polylog}(n))$, using $nH_k(T) + O(n/\log^{\varepsilon} n)$ bits. The data structure does not depend on the parameter k and the space bound holds simultaneously for all $k \leq \alpha \log_{|\Sigma|} n$ and constant $0 < \alpha < 1$. With our index, the counting of the occurrences of an arbitrary pattern $P[1, p]$ as a substring of T takes $O(p)$ time (i.e. no alphabet dependence). Locating each pattern occurrence takes $O(\log^{1+\varepsilon} n)$. Displaying a text substring of length ℓ takes $O(\ell + \log^{1+\varepsilon} n)$ time.

If the size of the alphabet is larger than polylogarithmic, that is, $|\Sigma| = O(n^{\beta})$ with $\beta < 1$, our data structure uses $nH_k(T) + o(n \log |\Sigma|)$ space and the query times are multiplied by a factor $O(\log |\Sigma| / \log \log n)$. Note that, although the space occupancy can now be $\Omega(n)$, it is still less than the size of the uncompressed text. Moreover, in terms of query time our index is faster than most FM-index implementations. As an alternative, if space is more important than query speed, we can use a simplified version of our index which uses binary wavelet trees instead of our new integer sequences representation. For any alphabet size, the space occupancy of this latter index is bounded by $nH_k(T) + O(\log |\Sigma| (n \log \log n / \log n))$ bits for any $k \leq \alpha \log_{|\Sigma|} n$, and $0 < \alpha < 1$. The index takes $O(p \log |\Sigma|)$ time to count the pattern occurrences, and $O(\log |\Sigma| \, (\ell + \log^2 n / \log \log n))$ time to locate and display a substring of length ℓ.

There exist several other compressed full-text indexes not based on the FM-index concept [8, 20, 38, 35, 18, 19, 29, 26]. Among them, the data structure with the smallest space occupancy is described in [18] (Theorems 4.2 and 5.2) and uses $nH_k(T) + O(\log |\Sigma| (n \log \log n / \log n))$ bits of storage for *a fixed* $k \leq \alpha \log_{|\Sigma|} n$ with $0 < \alpha < 1$ (the parameter k must be chosen when the index is built). The index in [18] takes $O(p \log |\Sigma| + \mathrm{polylog}(n))$ time to count the pattern occurrences and $O(\log |\Sigma| \, (\ell + \log^2 n / \log \log n))$ time to locate and display a substring of length ℓ. Note that for alphabets of polylogarithmic size our index is faster in both queries at the cost of a small increase in the big-Oh terms of the space occupancy. Note also that for any alphabet size our simplified data structure takes the same space as the index in [18], but it is faster in counting the occurrences and takes the same time in reporting and displaying the occurrences.

Finally, we point out that the structure in [18] uses binary wavelet trees to compactly represent sequences. By replacing binary wavelet trees with the sequence representation described in this paper we can remove the $O(\log |\Sigma|)$ factors from their query times.

To summarize, our index has asymptotically the smallest known space occupancy and processes all queries faster than the data structure in [18], which is the only other compressed index known to date with essentially $nH_k(T)$ space occupancy. Table 1 summarizes our contribution.

Reference	Space in bits	Counting time	Works for $\|\Sigma\| =$
[8, 9]	$5nH_k(T) + o(n)$	$O(p)$	$O(1)$
[39]	$nH_0(T) + O(n)$	$O(p)$	$O(\mathrm{polylog}(n))$
[28]	$2nH_k(T)\log\|\Sigma\| + O(n)$	$O(p)$	$O(\mathrm{polylog}(n))$
[38]	$nH_0(T) + O(n\log\log\|\Sigma\|)$	$O(p\log n)$	$o(n/\log n)$
[17]	$2n(H_0(T)+1)(1+o(1))$	$O(p\log\|\Sigma\|)$	$o(n/\log n)$
[18, 19]	$nH_k(T) + o(n\log\|\Sigma\|)$	$O(p\log\|\Sigma\| + \mathrm{polylog}(n))$	$o(n/\log n)$
[35]	$4nH_k(T) + o(n)$	$O(p^3\log\|\Sigma\| + p\log n)$	$o(n/\log n)$
This paper	$nH_k(T) + o(n)$	$O(p)$	$O(\mathrm{polylog}(n))$
	$nH_k(T) + o(n\log\|\Sigma\|)$	$O(p\ \log\|\Sigma\|/\log\log n)$	$O(n^\beta)$, $\beta < 1$
	$nH_k(T) + o(n\log\|\Sigma\|)$	$O(p\ \log\|\Sigma\|)$	$o(n/\log n)$

Table 1. Comparison of space, counting time, and restrictions on the alphabet size among the best known self-indexes.

3 Compressed Representation of Sequences

Let $S = s_1s_2\dots s_n$ be a sequence of n integers in the range $[1, r]$, called *symbols* from now on. In this section we face the problem of supporting Rank and Select queries on S. The query $\mathsf{Rank}_c(S, q)$ returns the number of times symbol $c \in [1, r]$ appears in $S[1, q] = s_1s_2\dots s_q$. The query $\mathsf{Select}_c(S, q)$ returns the position in S of the q-th occurrence of symbol $c \in [1, r]$. Our aim is to represent S in compressed form, and hence we need to support also the retrieval of any $S[q] = s_q$.

We measure the size of the compressed representation of S as a function of its *zero-order empirical entropy* $H_0(S) = -\sum_c (n_c/n)\log(n_c/n)$, where n_c is the number of occurrences of symbol c in S, $n = \sum_c n_c = |S|$, and all logarithms are taken to the base 2 (with $0\log 0 = 0$). Notice that $nH_0(S)$ is an Information Theoretic lower bound to the compressibility of S when we use a fixed codeword for each of its symbols.

3.1 Representing Sequences of Small Integers

In this section we describe a first data structure which represents S in $nH_0(S)+O((rn\log\log n)/\log_r n)$ bits, and answers $S[q]$, $\mathsf{Rank}_c(S, q)$ and $\mathsf{Select}_c(S, q)$ queries in constant time. For the construction of the data structure we only need $2 \le r \le \sqrt{n}$, but the data structure is interesting only for $r = o(\log n/\log\log n)$ since otherwise the space occupancy will exceed the space $\Omega(n\log r)$ used by the standard uncompressed representation. We first focus on $S[q]$ and $\mathsf{Rank}_c(S, q)$ queries, and address the $\mathsf{Select}_c(S, q)$ query later.

Structure. We divide S into blocks of size $u = \left\lfloor \frac{1}{2}\log_r n \right\rfloor$. Consider a block where each symbol c appears N^c times, so $N^1 + N^2 + \dots + N^r = u$. We say that tuple $(N^1, N^2, \dots, N^r)$ is the *symbol composition* of the block. Using this notation, we define the following sequences of values indexed by block number $i = 1, \dots, \lceil n/u \rceil$:

- $S_i = S[u(i-1)+1\dots ui]$ is the sequence of symbols forming the i-th block of S.
- For each symbol $c \in [1, r]$, $N_i^c = \mathsf{Rank}_c(S_i, u)$ is the number of occurrences of c in S_i.
- $L_i = \left\lceil \log \binom{u}{N_i^1,\dots,N_i^r} \right\rceil$ is the number of bits necessary to encode all the possible sequences of u symbols in $[1, r]$ that share the symbol composition $(N_i^1, \dots, N_i^r)$ of block S_i.
- I_i is the identifier of block S_i among all sequences having its symbol composition $(N_i^1, \dots, N_i^r)$. I_i consists of L_i bits.

– R_i is the identifier of the symbol composition $(N_i^1, \ldots, N_i^r)$ among all possible tuples of r numbers that add up to u.[1] R_i consists of $\left\lceil \log \binom{u+r-1}{r-1} \right\rceil \leq \lceil r \log(u+1) \rceil$ bits.

Our compressed representation of S consists of the storage of the following information:

– A bit sequence I obtained by concatenating all variable-length identifiers I_i.
– A bit sequence R obtained by concatenating all fixed-length identifiers R_i.
– A table $E = E_{N^1,\ldots,N^r}$ for every possible symbol composition $(N^1, \ldots, N^r)$ summing up to u. Each entry of E corresponds to a different u-length block G (with the proper symbol composition $(N^1, \ldots, N^r)$) and stores the answers to all $\mathsf{Rank}_c(G, q)$ queries, where $1 \leq q \leq u$ and $c \in [1, r]$. Indexes I_i are such that $E[I_i]$ stores Rank_c-information for the block S_i. Tables E do not depend on S, but just on u.
– A table F whose entries are indexed by all possible symbol compositions $(N^1, \ldots, N^r)$ summing up to u, and point to the corresponding tables $E_{N^1,\ldots,N^r}$. Indexes R_i are such that $F[R_i]$ points to $E_{N_i^1,\ldots,N_i^r}$. Also table F does not depend on S, but just on u.
– Information to answer *partial sum* queries on L_i, that is, to compute $\sum_{j=1}^{i} L_j$ in constant time for any block i.
– Information to answer partial sum queries on N_i^c, that is, to compute $\sum_{j=1}^{i} N_j^c$ in constant time for any block i and symbol c.

Solving queries. To answer queries about position q we first compute the block number $i = \lceil q/u \rceil$ to which q belongs and the offset $\ell = q - (i-1)u$ inside that block. Then we compute $E = F[R_i]$, the table of entries corresponding to block i, and $G = E[I_i]$, the entry of E corresponding to block i. Note that, since the I_i values use variable number of bits, we need to know which is the starting and ending positions of the representation for I_i in the sequence. These are $1 + \sum_{j=1}^{i-1} L_j$ and $\sum_{j=1}^{i} L_j$, respectively, which are known in constant time because we have partial sum information on L_i.

Now, to answer $\mathsf{Rank}_c(S, q)$ we evaluate (in constant time) the partial sum $\sum_{j=1}^{i-1} N_j^c$ and add $\mathsf{Rank}_c(G, \ell)$. To answer $S[q]$ we simply give $G[\ell]$. Both queries take constant time. Figure 1 graphically illustrates the *rank* computation.

Space usage. First notice that $uH_0(S_i) = \log \binom{u}{N_i^1,\ldots,N_i^r}$, and thus

$$\begin{aligned}\sum_{i=1}^{\lceil n/u \rceil} uH_0(S_i) &= \sum_{i=1}^{\lceil n/u \rceil} \log \binom{u}{N_i^1, \ldots, N_i^r} = \log \prod_{i=1}^{\lceil n/u \rceil} \binom{u}{N_i^1, \ldots, N_i^r} \\ &\leq \log \binom{n}{n_1, \ldots, n_r} = nH_0(S), \end{aligned} \qquad (1)$$

where n_c is the total number of occurrences of character c in S. The inequality holds because distributing N_i^c symbols over block S_i is just one possible way to distribute n_c symbols over S [36]. This result permits us to bound the length of the sequence I as

$$\begin{aligned}\sum_{i=1}^{\lceil n/u \rceil} L_i &= \sum_{i=1}^{\lceil n/u \rceil} \left\lceil \log \binom{u}{N_i^1, \ldots, N_i^r} \right\rceil \leq \sum_{i=1}^{\lceil n/u \rceil} uH_0(S_i) + \lceil n/u \rceil \\ &\leq nH_0(S) + O(n/\log_r n).\end{aligned}$$

[1] In the binary case ($r = 2$), R_i is just the number of bits set in S_i [37], but this is more complicated here.

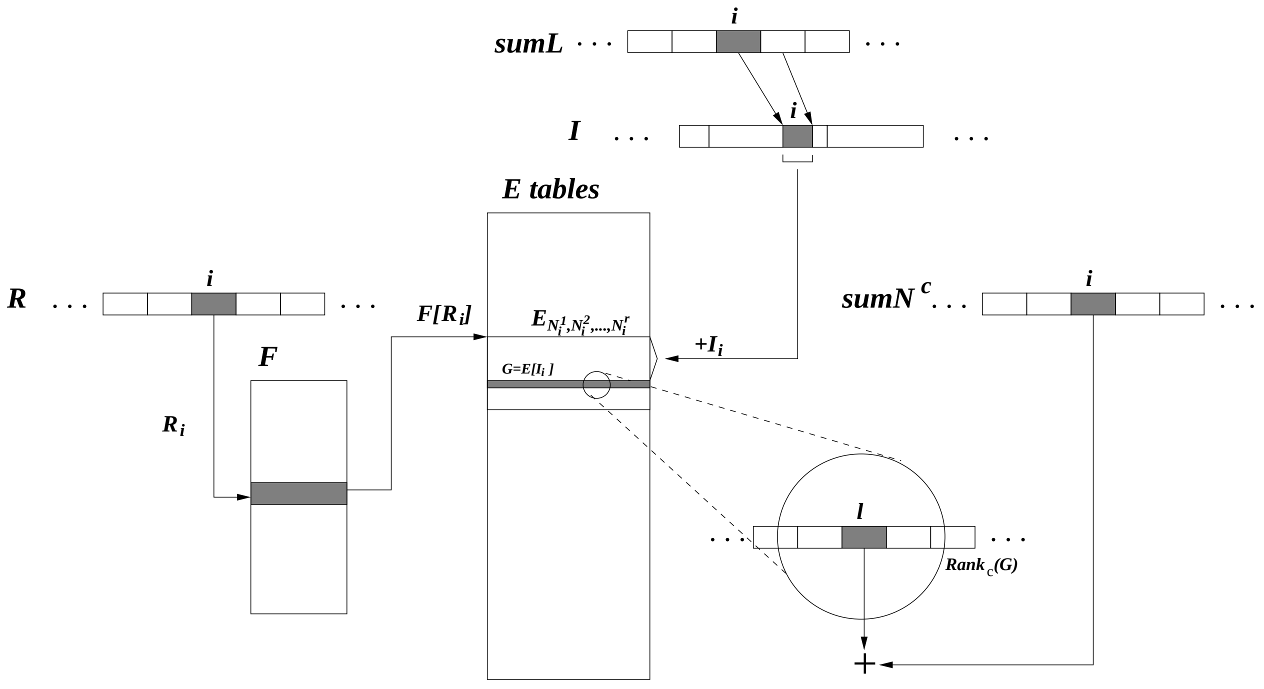

Fig. 1. A graphical description of the algorithm solving *rank* queries on sequences. Precisely, we illustrate the case $\mathsf{Rank}_c(S,q)$, where $q = i \cdot u + l$. The final "+" is the answer.

Let us now consider the sequence R. The number of different tuples $(N^1,\ldots,N^r)$ that add up u is $\binom{u+r-1}{r-1} \le (u+1)^r$. Hence it is enough to use $\lceil r\log(u+1)\rceil$ bits for each R_i (which actually is enough to describe any tuple of r numbers in $[0,u]$). Accumulated over the $\lceil n/u\rceil$ blocks, this requires $O(rn\log\log n/\log_r n)$ bits.

We consider now the structures to answer partial sum queries [36], namely $\sum_{j=1}^{i} N_j^c$ and $\sum_{j=1}^{i} L_j$. Both structures are similar. Let us first concentrate on the L_j's, whose upper bounds are $L_i \le \lceil u\log r\rceil$, since $\binom{u}{N_i^1,\ldots,N_i^r} \le r^u$. Recall that we need to answer partial sum queries over the sequence of integers $L = L_1, L_2, \ldots, L_t$, where $t = \lceil n/u\rceil$. Since $L_i \le \lceil u\log r\rceil$, each partial sum over L does not exceed $n\lceil\log r\rceil$ and can be represented in $\lceil\log(n\lceil\log r\rceil)\rceil$ bits. Divide L into blocks of that length, $\lceil\log(n\lceil\log r\rceil)\rceil$, and store the full partial sums for each block beginning. This requires exactly $t = O(n/\log_r n)$ bits. Inside each block, store the partial sums relative to the block beginning. These latter partial sums cannot exceed $\lceil u\log r\rceil\lceil\log(n\lceil\log r\rceil)\rceil$ because of the upper bound on the L_i's and the length of the L-blocks. Hence we need $O(\log u + \log\log r + \log\log n) = O(\log\log n)$ bits for each partial sum within each block of L. Thus, we need $O(|L|\log\log n) = O(t\log\log n) = O(n\log\log n/\log_r n)$ bits overall for the partial sum information on L. A partial sum query on L_i is answered in constant time by adding the partial sum of the block of L that contains L_i and the relative partial sum of L_i inside that block. The same technique can be applied to sequences N^c, whose values are in the range $[0,u]$, to obtain $O(rn\log\log n/\log_r n)$ bits of space because there are r sequences to index.

Finally, let us consider tables E and F. The total number of entries over all $E_{N^1,\ldots,N^r}$ tables is clearly r^u since each sequence of u symbols over $[1,r]$ belongs exactly to one symbol composition. For each such entry G we explicitly store the sequence itself plus the answers to all $\mathsf{Rank}_c(G,q)$

queries, $c \in [1, r]$, $1 \leq q \leq u$. This storage requires $O(u \log r + ru \log u)$ bits. Added over all the entries of all the E tables, we have $O(r^u(u \log r + ru \log u)) = O(\sqrt{n} r \log_r n \log \log n)$ bits, which is $o(rn \log \log n / \log_r n)$. Table F has necessarily less entries than E, since there is at least one distinct entry of E for each $(N^1, \ldots, N^r)$ symbol composition in F. Each entry in F points to the corresponding table $E_{N^1,\ldots,N^r}$. If we concatenate all $E_{N^1,\ldots,N^r}$ tables into a supertable of r^u entries, then F points inside that supertable, to the first entry of the corresponding table, and this needs $O(u \log r)$ bits per entry. Overall this adds $O(r^u u \log r)$ bits, which is negligible compared to the size of E.

We remark that the simpler solution of storing indexes $P_i = F[R_i] + I_i$ directly pointing to the large supertable E would require $n \log r$ bits as the pointers are as long as the sequences represented. This would defeat the whole scheme. Thus we use table F as an intermediary so as to store the smaller R_i (subtable identifier) and I_i (index relative to the subtable).

Solving $\mathsf{Select}_c(S, q)$ queries. The solution to $\mathsf{Select}_c(S, q)$ queries on binary sequences proposed in [37, Lemma 2.3] divides the sequence into blocks of size u (with the same formula we use for u, with $r = 2$) and makes use of a sequence A, so that A_i is the number of bits set in the i-th block. In our scheme, sequence A corresponds to sequence N^c for each character $c \in [1, r]$. We can use exactly the same scheme of [37] for each of our sequences N^c. They need precisely the same partial sum queries we already considered for N^c, as well as other structures that require $O(n \log(u)/u)$ bits per sequence N^c. They also need to have all $\mathsf{Select}_c(G, q)$ queries precomputed for each possible block G, which we can add to our E tables for additional $O(r^u ru \log u)$ bits. Overall, the solution needs $O(rn \log(u)/u) = O(rn \log \log n / \log_r n)$ additional bits of space.

Theorem 1. *Let $S[1, n]$ be a sequence of numbers in $[1, r]$, with $2 \leq r \leq \sqrt{n}$. There exists a data structure using $nH_0(S) + O(r(n \log \log n)/\log_r n)$ bits of space, that supports queries $\mathsf{Rank}_c(S, q)$ and $\mathsf{Select}_c(S, q)$, and retrieves $S[q]$, all in constant time.* □

The theorem is a generalization of the result in [36, 37], which uses $nH_0(S) + O((n \log \log n)/\log n)$ bits of space to represent a *binary* sequence S $(r = 2)$ so as to answer $\mathsf{Rank}_c(S, q)$ and $\mathsf{Select}_c(S, q)$ queries in constant time. Note that for binary sequences queries $S[q]$ can be easily answered in constant time by finding $c \in \{0, 1\}$ such that $\mathsf{Rank}_c(S, q) - \mathsf{Rank}_c(S, q - 1) = 1$, whereas we had to provide direct access to $G[\ell]$. Moreover, with binary sequences one can use $R_i = i$, while we needed explicit pointers to an intermediate table F.

As we have already observed, the above result is interesting only for alphabets of size $r = o(\log n / \log \log n)$, since otherwise the space occupancy of the data structure is $\Omega(n \log r)$. In the next section we show how to extend this result to alphabets of polylogarithmic size.

3.2 Generalized Wavelet Trees

In this section we use the representation of sequences developed in the previous section to build an improved sequence representation, which adapts better to the range of different symbols represented. Albeit we solve exactly the same problem, we will change notation a little bit for clarity. This time our sequence $S[1, n]$ will be a sequence of symbols over an *alphabet* $\Sigma = [1, |\Sigma|]$, so that $r \leq |\Sigma|$ will be reserved to applications of Theorem 1. Actually, for $r = |\Sigma|$ the data structure we are going to introduce will be essentially the data structure of Theorem 1.

Let us recall the basic ideas underlying the (binary) wavelet tree [18]. Consider a balanced binary tree $\mathcal{T}$ whose leaves contain the characters of the alphabet Σ. $\mathcal{T}$ has height $O(\log |\Sigma|)$. Each node u of $\mathcal{T}$ is associated with a string S_u that represents the subsequence of S containing *only* the characters that are stored into leaves descending from u. The root is thus associated with the entire S. The node u of the wavelet tree does store indeed a *binary image* of the sequence S_u, denoted by B_u, such that $B_u[i] = 1$ if the character $S_u[i]$ descends from the *right* child of u, and $B_u[i] = 0$ otherwise. By representing every binary sequence B_u with the data structure of [37] we get a data structure supporting $\mathsf{Rank}_c(S, q)$, $\mathsf{Select}_c(S, q)$, and $S[q]$ queries in $O(\log |\Sigma|)$ time using $nH_0(S) + o(n)$ space.

The r-ary Wavelet Tree. Let us consider an *r-ary balanced tree* whose leaves are associated with the symbols of the alphabet Σ. This r-ary tree has height at most $1 + \log_r |\Sigma|$. As in a (binary) wavelet tree, each node v is associated with a subsequence S_v of $S[1, n]$ formed just by the symbols descending from v. Unlike a (binary) wavelet tree, the subsequence S_v is stored in v as a *sequence of integers in the range* $[1, r]$. Precisely, let v be a node with children $v_1 \ldots v_r$, and let Σ_v be the set of symbols descending from v. Because of the balancedness of the tree, Σ_v is actually split into r equally-sized subsets $\Sigma_{v_1} \ldots \Sigma_{v_r}$, which are integral ranges of size $|\Sigma_{v_i}| \approx |\Sigma_v|/r$. Therefore, the sequence S_v is represented as a sequence of $n_v = |S_v|$ integers in the range $[1, r]$ such that $S_v[q] = j$ whenever $S_v[q] \in \Sigma_{v_j}$. The data structure of Theorem 1 is finally built over S_v and stored at node v so to answer queries $S_v[q]$, $\mathsf{Rank}_j(S_v, q)$, and $\mathsf{Select}_j(S_v, q)$ in constant time.

A technical detail is that we concatenate all the sequences S_v lying at the tree level h, and store them into one unique long sequence S^h. All these level-wise sequences have the same length of S, namely n. As we go down/up the tree, it is easy to maintain in constant time the index $q^* + 1$ where the current node sequence S_v starts inside the level sequence S^h. To achieve this, each node v maintains also a vector $C_v[1, r]$ such that $C_v[j]$ is the number of occurrences in S_v of symbols in $[1, j-1]$. Now assume that we are at node v and its sequence S_v starts at index $q^* + 1$ of S^h; then the sequence S_{v_j} of the jth child of v starts at $q^* + C_v[j] + 1$ in S^{h+1}. Conversely, assume that we are at node v_j and its sequence S_{v_j} starts at index $q^* + 1$ of S^{h+1}; then the sequence S_v of the parent v of v_j starts at $q^* - C_v[j] + 1$ in S^h. Notice that we need to store pointers (with negligible extra space) to find the C vectors of children or parents, or we can take advantage of the tree being almost perfect to avoid such pointers. We need also, for the bottom-up traversal required to implement *select* (see next), $|\Sigma|$ pointers to the leaves of the r-ary wavelet tree.

Solving queries. To compute $\mathsf{Rank}_c(S, q)$, we start at the root node v and determine in constant time the subset Σ_{v_j} to which c belongs by a simple algebraic calculation. We then compute the position corresponding to q in S_{v_j}, namely $q_{v_j} = \mathsf{Rank}_j(S_v, q)$. We then recursively continue with $q = q_{v_j}$ at node v_j. We eventually reach a tree leaf v_l (corresponding to the subset $\{c\} \subseteq \Sigma$), for which we have the answer to our original query $\mathsf{Rank}_c(S, q) = q_{v_l}$. On the other hand, to determine $S[q]$, we start at the root node v and obtain $j = S_v[q]$, so that $S[q] \in \Sigma_{v_j}$. Then we continue recursively with node v_j and $q = q_{v_j} = \mathsf{Rank}_j(S_v, q)$ as before, until we reach a leaf v_l, where $\Sigma_{v_l} = \{S[q]\}$ is finally determined. Both queries take $O(\log_r |\Sigma|)$ time.

To compute $\mathsf{Select}_c(S, q)$, instead, we proceed bottom-up. We identify the leaf v_l corresponding to subset $\{c\}$ and then proceed upwards. At leaf v_l (not actually represented in the tree), we initialize $q_{v_l} = q$. This is the position we want to track upwards in the tree. Now, let v be the parent of v_j, then $q_v = \mathsf{Select}_j(S_v, q_{v_j})$ is the position of $S_{v_j}[q_{v_j}]$ in S_v. We eventually reach the root, with $q_{root} = \mathsf{Select}_c(S, q)$, in $O(\log_r |\Sigma|)$ time.

It goes without saying that, since we do not represent sequences S_v but level-wise sequences S^h, in the calculations above we need to take care of the latter. Assume that our current sequence S_v starts at position $q^* + 1$ in S^h. Then, queries over S_v are translated to queries over S^h as follows: $S_v[q] = S^h[q^* + q]$, $\mathsf{Rank}_j(S_v, q) = \mathsf{Rank}_j(S^h, q^* + q) - \mathsf{Rank}_j(S^h, q^*)$, and $\mathsf{Select}_j(S_v, q) = \mathsf{Select}_j(S^h, \mathsf{Rank}_j(S^h, q^*) + q)$.

Space usage. An immediate advantage of having all sequences $S^h[1, n]$ over the same alphabet $[1, r]$ is that all tables E and F are the same for all levels, so they take $o((rn \log\log n)/\log_r n)$ bits overall. All the other $O((rn \log\log n)/\log_r n)$ size structures used to prove Theorem 1 totalize $O(\log|\Sigma|\,(rn \log\log n)/\log n)$ bits of space by adding up all the $O(\log_r |\Sigma|)$ levels. The structures C_v need $O(r \log n)$ bits each, and there is one C_v array per non-leaf node v. This totalizes $O\left(\frac{|\Sigma|}{r-1} r \log n\right) = O(|\Sigma| \log n)$ bits. This space includes also the pointers to leaves and the parent-child pointers in the tree, if they are used.

Let us consider now the entropy-related part. For each non-leaf node v at tree level h, with children $v_1, \ldots, v_r$, sequence S_v spans at most $2 + \lfloor n_v/u \rfloor$ blocks in S^h (recall from Section 3.1 that the sequence is divided into blocks of length $u = \left\lfloor \frac{1}{2} \log n \right\rfloor$ and that $n_v = |S_v|$). The sum of local zero-order entropies $uH_0(S_i^h)$ for the $\lfloor n_v/u \rfloor$ blocks is a lower bound to $n_v H_0(S_v)$ (recall Eq. (1)). For the other 2 blocks, we simply assume that they take the maximum $u \lceil \log r \rceil = O(\log n)$ bits. We have at most $\frac{r}{r-1}|\Sigma|$ boundaries over the whole tree. Hence summing over all the sequence boundaries, the space overhead induced by all the partial blocks is $O(|\Sigma| \log n)$ bits.

Thus, let us focus on the term $n_v H_0(S_v)$. Note that this is

$$\begin{aligned} -n_v \sum_{j=1}^{r} \frac{n_{v_j}}{n_v} \log \frac{n_{v_j}}{n_v} &= -\sum_{j=1}^{r} n_{v_j} \log n_{v_j} + \sum_{j=1}^{r} n_{v_j} \log n_v \\ &= n_v \log n_v - \sum_{j=1}^{r} n_{v_j} \log n_{v_j}. \end{aligned}$$

If we add this term over all the nodes v in the tree, we get a sort of telescopic sum in which the second terms $-n_{v_j} \log n_{v_j}$ computed for v will cancel with the first (positive) term of the formula derived for the children v_j's. Therefore, after all the cancellations, the only surviving terms are: the term $n \log n$ corresponding to the tree root, and the terms $-n_{u_l} \log n_{u_l}$ corresponding to the parents of the tree leaves (where $n_{u_l} = n_c$ for some $c \in \Sigma$, being n_c the frequency of character c). This is

$$n \log n - \sum_{c \in \Sigma} n_c \log(n_c) \quad = \quad nH_0(S).$$

Theorem 2. *Let $S[1, n]$ be a string over an arbitrary alphabet Σ. The r-ary wavelet tree built on S, for $2 \leq r \leq \min(|\Sigma|, \sqrt{n})$, uses $nH_0(S) + O(|\Sigma| \log n) + O(\log|\Sigma|\,(rn \log\log n)/\log n)$ bits of storage and supports in $O(\log_r |\Sigma|)$ time the queries $S[q]$, $\mathsf{Rank}_c(S, q)$ and $\mathsf{Select}_c(S, q)$, for any $c \in \Sigma$ and $1 \leq q \leq n$.*

Moreover, if $|\Sigma| = O(\mathrm{polylog}(n))$, then r can be chosen so that the resulting r-ary wavelet tree supports all queries in constant time and takes $nH_0(S) + O(n/\log^{\varepsilon} n)$ bits of space, for any constant $0 < \varepsilon < 1$.

Proof. The first part of the theorem, for a general r, is a consequence of the development in this section. For the last sentence, note that by choosing $r = |\Sigma|^{1/\kappa}$, for constant $\kappa > 0$, we can support the query operations in constant time $O(\kappa)$. Now, if $|\Sigma| = O(\text{polylog}(n)) = O((\log n)^d)$, then we can choose any $\kappa > d$ to obtain $O(dn(\log\log n)^2/(\log n)^{1-d/\kappa})$ space overhead. For any constant $0 < \varepsilon < 1$, we choose $d < \kappa < d/(1-\varepsilon)$ to ensure that $O(n(\log\log n)^2/(\log n)^{1-d/\kappa}) = O(n/\log^{\varepsilon} n)$. □

The theorem is a generalization upon the (binary) wavelet tree data structure [18], which takes $nH_0(S) + O(\log|\Sigma|\ (n\log\log n)/\log n)$ space and answers the same queries in $O(\log|\Sigma|)$ time. The last part shows that, when $|\Sigma| = O(\text{polylog}(n))$, the generalization obtains essentially the same space (up to lower order terms) and reduces query times to a constant. The case of larger alphabets deserves a separate corollary.

Corollary 1. *Let $S[1,n]$ be a string over an alphabet Σ. If we choose $r = O(\log n/(\log\log n)^2)$, the r-ary wavelet tree built on S uses $nH_0(S) + O(|\Sigma|\log n) + o(n\log|\Sigma|)$ bits and supports in $O(\log|\Sigma|/\log\log n)$ time the queries $S[q]$, $\mathsf{Rank}_c(S,q)$ and $\mathsf{Select}_c(S,q)$, for any $c \in \Sigma$ and $1 \le q \le n$. Note that if $|\Sigma| = O(n^{\beta})$, $\beta < 1$, the space occupancy simplifies to $nH_0(S) + o(n\log|\Sigma|)$.* □

4 Compressed Representation of Full-Text Indexes

We will now apply the results of the previous section to build a new implementation of the FM-index concept. We need first to explain the FM-index concept in full detail, that is, the Burrows-Wheeler transform and the backward search mechanism, highlighting the time dependence on the alphabet size $|\Sigma|$. Also, the compression boosting technique [10] will be central to our solution. We introduce these previous developments in Sections 4.1 to 4.4 and then introduce our new result in Section 4.5.

Hereafter we assume that $T[1,n]$ is the text we wish to index, compress and query. T is drawn from an alphabet Σ of size $|\Sigma|$. By $T[i]$ we denote the i-th character of T, $T[i,n]$ denotes the ith text suffix, and $T[1,i]$ denotes the ith text prefix. A substring of T is any $T[i,j]$. We write $|w|$ to denote the length of string w.

Our final goal is to answer *substring queries* using a compressed representation of T. That is, we wish to find out whether a given *pattern* $P[1,p]$ occurs as a substring of T (existence query), how many times it occurs (counting query), and at which positions (locating query). Also, as T is compressed we need to support the retrieval of any substring of T (context query). If the compressed representation of T supports all these queries, we say that the representation is a *compressed full-text self-index*.

4.1 The k-th Order Empirical Entropy

Following a well established practice in Information Theory, we lower bound the space needed to store a string T by using the notion of *empirical entropy*. The empirical entropy is similar to the entropy defined in the probabilistic setting with the difference that it is defined in terms of the character frequencies observed in T rather than in terms of character probabilities. The key property of empirical entropy is that it is defined *pointwise* for *any* string T and can be used to measure the performance of compression algorithms as a function of the *string structure*, thus *without* any assumption on the input source. In a sense, compression bounds produced in terms of empirical entropy are *worst-case measures*.

Just as defined for sequences, the *zero-order* empirical entropy of T is defined as $H_0(T) = -\sum_c (n_c/n)\log(n_c/n)$, where n_c is the number of occurrences of alphabet character c in T, and

		F		T^{bwt}
`mississippi#`		`#`	`mississipp`	`i`
`ississippi#m`		`i`	`#mississip`	`p`
`ssissippi#mi`		`i`	`ppi#missis`	`s`
`sissippi#mis`		`i`	`ssippi#mis`	`s`
`issippi#miss`		`i`	`ssissippi#`	`m`
`ssippi#missi`	$\Longrightarrow$	`m`	`ississippi`	`#`
`sippi#missis`		`p`	`i#mississi`	`p`
`ippi#mississ`		`p`	`pi#mississ`	`i`
`ppi#mississi`		`s`	`ippi#missi`	`s`
`pi#mississip`		`s`	`issippi#mi`	`s`
`i#mississipp`		`s`	`sippi#miss`	`i`
`#mississippi`		`s`	`sissippi#m`	`i`

Fig. 2. Example of Burrows-Wheeler transform for the string $T =$ `mississippi`. The matrix on the right has the rows sorted in lexicographic order. The output of the BWT is the last column; in this example the string `ipssm#pissii`.

$n = \sum_c n_c = |T|$. To introduce the concept of *k-th order* empirical entropy we need to define what is a *context*. A length-k context w in T is one of its substrings of length k. Given w, we denote by w_T the string formed by concatenating all the symbols following the occurrences of w in T, taken from left to right. For example, if $T =$ `mississippi` then $\mathtt{s_T} =$ `sisi` and $\mathtt{si_T} =$ `sp`. The k-th order empirical entropy of T is defined as:

$$H_k(T) = \frac{1}{n} \sum_{w \in \Sigma^k} |w_T| \, H_0(w_T). \tag{2}$$

The k-th order empirical entropy $H_k(T)$ is a lower bound to the output size of any compressor which encodes each character of T using a uniquely decipherable code that depends only on the character itself and on the k characters preceding it. For any $k \geq 0$ we have $H_k(T) \leq \log |\Sigma|$. Note that for strings with many regularities we may have $H_k(T) = o(1)$. This is unlike the entropy defined in the probabilistic setting which is always a constant. As an example, for the family of texts $T = (ab)^{n/2}$ we have $H_0(T) = 1$ and $H_k(T) = O((\log n)/n)$ for any $k \geq 1$.

4.2 The Burrows-Wheeler Transform

In [1] Burrows and Wheeler introduced a new compression algorithm based on a reversible transformation now called the *Burrows-Wheeler Transform* (BWT from now on). The BWT consists of three basic steps (see Fig. 2): (1) append at the end of T a special character # smaller than any other text character; (2) form a *conceptual* matrix $\mathcal{M}_T$ whose rows are the cyclic shifts of the string $T\#$ sorted in lexicographic order; (3) construct the transformed text T^{bwt} by taking the last column of matrix $\mathcal{M}_T$. Notice that every column of $\mathcal{M}_T$, hence also the transformed text T^{bwt}, is a permutation of $T\#$. In particular the first column of $\mathcal{M}_T$, call it F, is obtained by lexicographically sorting the characters of $T\#$ (or, equally, the characters of T^{bwt}).

We remark that the BWT by itself is not a compression algorithm since T^{bwt} is just a permutation of $T\#$. However, if T has "small" entropy the transformed string T^{bwt} contains long runs of identical characters and turns out to be highly compressible (see [1, 31] for details).

Because of the special character #, when we sort the rows of $\mathcal{M}_T$ we are essentially sorting the suffixes of T. Therefore there is a strong relation between the matrix $\mathcal{M}_T$ and the suffix array built on T. The matrix $\mathcal{M}_T$ has also other remarkable properties; to illustrate them we introduce the following notation:

- Let $C[\,]$ denote the array of length $|\Sigma|$ such that $C[c]$ contains the total number of text characters which are alphabetically smaller than c.
- Let $\mathsf{Occ}(c, q)$ denote the number of occurrences of character c in the prefix $T^{bwt}[1, q]$. In our sequence terminology, $\mathsf{Occ}(c, q) = \mathsf{Rank}_c(T^{bwt}, q)$.
- Let $LF(i) = C[T^{bwt}[i]] + \mathsf{Occ}(T^{bwt}[i], i)$.

$LF(\ )$ stands for *Last-to-First* column mapping since the character $T^{bwt}[i]$, in the last column of $\mathcal{M}_T$, is located in the first column F at position $LF(i)$. For example in Fig. 2 we have $LF(10) = C[\mathtt{s}] + \mathsf{Occ}(\mathtt{s}, 10) = 12$; and in fact $T^{bwt}[10]$ and $F[LF(10)] = F[12]$ both correspond to the first `s` in the string `mississippi`.

The $LF(\ )$ mapping allows us to scan the text T backward. Namely, if $T[k] = T^{bwt}[i]$ then $T[k-1] = T^{bwt}[LF(i)]$. For example in Fig. 2 we have that $T[3] = \mathtt{s}$ is the 10th character of T^{bwt} and we correctly have $T[2] = T^{bwt}[LF(10)] = T^{bwt}[12] = \mathtt{i}$ (see [8] for details).

4.3 The FM-Index

The FM-index is a self-index that allows one to efficiently search for the occurrences of an arbitrary pattern $P[1, p]$ as a substring of the text $T[1, n]$. Pattern P is provided on-line whereas the text T is given to be preprocessed in advance. The number of pattern occurrences in T is hereafter indicated with *occ*.

The FM-index consists, in essence, of the data structures required to compute $C[\ ]$, $\mathsf{Occ}(\ )$, and $LF(\ )$. The first is directly stored in $|\Sigma| \log n$ bits. To compute $\mathsf{Occ}(c, q)$ in constant time, the FM-index stores a compressed representation of T^{bwt} together with some auxiliary information. This also gives the tools to compute $LF(i)$, provided we have access to $T^{bwt}[i]$. This is obtained in $O(|\Sigma|)$ time by linearly searching for the only $c \in \Sigma$ such that $\mathsf{Occ}(c, q) \neq \mathsf{Occ}(c, q-1)$. The two key procedures to operate on the FM-index are: the *counting* of the number of pattern occurrences (shortly count), and the *location* of their positions in the text T (shortly locate). Note that the counting process returns the value *occ*, whereas the location process returns *occ* distinct integers in the range $[1, n]$.

```
Algorithm count($P[1,p]$)
1. $i \leftarrow p$, $c \leftarrow P[p]$, First $\leftarrow 1$, Last $\leftarrow n$;
2. while ((First $\leq$ Last) and ($i \geq 1$)) do
3.     $c \leftarrow P[i]$;
4.     First $\leftarrow C[c] + \mathsf{Occ}(c, \mathsf{First} - 1) + 1$;
5.     Last $\leftarrow C[c] + \mathsf{Occ}(c, \mathsf{Last})$;
6.     $i \leftarrow i - 1$;
7. if (Last < First) then return "no rows prefixed by $P[1,p]$" else return (First, Last).
```

Fig. 3. Algorithm count for finding the set of rows prefixed by $P[1, p]$, and thus for counting the pattern occurrences $occ = \mathsf{Last} - \mathsf{First} + 1$. Recall that $C[c]$ is the number of text characters which are alphabetically smaller than c, and that $\mathsf{Occ}(c, q)$ denotes the number of occurrences of character c in $T^{bwt}[1, q]$.

Fig. 3 sketches the pseudocode of the counting operation that works "backwards" in p phases, hence numbered from p to 1. The i-th phase preserves the following invariant: *The parameter*

First *points to the first row of the BWT matrix* $\mathcal{M}_T$ *prefixed by* $P[i,p]$, *and the parameter* Last *points to the last row of* $\mathcal{M}_T$ *prefixed by* $P[i,p]$. After the final phase, P prefixes the rows between First and Last and thus, according to the properties of matrix $\mathcal{M}_T$ (see Section 4.2), we have $occ = \mathsf{Last} - \mathsf{First} + 1$. It is easy to see that the running time of count is dominated by the cost of the $2p$ computations of the values $\mathsf{Occ}(\ )$.

```
Algorithm locate(i)

1. i' ← i, t ← 0;
2. while row i' is not marked do
3.     i' ← LF[i'];
4.     t ← t + 1;
5. return Pos(i') + t;
```

Fig. 4. Algorithm locate for the computation of $\mathsf{Pos}(i)$.

Given the range $(\mathsf{First}, \mathsf{Last})$, we now consider the problem of locating the positions in T of these pattern occurrences. We notice that every row in $\mathcal{M}_T$ is prefixed by some suffix of T. For example, in Fig. 2 the fourth row of $\mathcal{M}_T$ is prefixed by the text suffix $T[5,11] =$ `issippi`. Then, for $i = \mathsf{First}, \mathsf{First}+1, \ldots, \mathsf{Last}$ we use procedure $\mathsf{locate}(i)$ to find the starting position in T of the suffix that prefixes the i-th row $\mathcal{M}_T[i]$. Such a position is denoted hereafter by $\mathsf{Pos}(i)$, and the pseudocode of locate is given in Fig. 4. The intuition underlying its functioning is simple. We scan backward the text T using the $LF(\ )$ mapping (see Section 4.2) until a *marked* position is met. If we mark one text position every $\Theta(\log^{1+\varepsilon} n)$, for some constant $\varepsilon > 0$, the `while` loop is executed $O(\log^{1+\varepsilon} n)$ times. Since the computation of $LF(i)$ is done via at most $|\Sigma|$ computations of $\mathsf{Occ}(\ )$, we have that locate takes $O(|\Sigma| \ \log^{1+\varepsilon} n)$ time. The space required by the marked positions is $\Theta(n/\log^{\varepsilon} n)$ bits. Combining the observations on locate with the ones for count, we get [8]:

Theorem 3. *For any string* $T[1,n]$ *drawn from a constant-sized alphabet* Σ, *the FM-index counts the occurrences of any pattern* $P[1,p]$ *within* T *taking* $O(p)$ *time. The location of each pattern occurrence takes* $O(|\Sigma| \ \log^{1+\varepsilon} n)$ *time, for any constant* $\varepsilon > 0$. *The size of the FM-index is bounded by* $5nH_k(T) + o(n)$ *bits, for any fixed* k. □

In order to retrieve the content of $T[l_1, l_2]$, we must first find the row in $\mathcal{M}_T$ that corresponds to l_2, and then issue $\ell = l_2 - l_1 + 1$ backward steps in T, using the $LF(\ )$ mapping. Starting at the lowest marked text position that follows l_2, we perform $O(\log^{1+\varepsilon} n)$ steps until reaching l_2. Then, we perform ℓ additional LF-steps to collect the text characters. The resulting complexity is $O((\ell + \log^{1+\varepsilon} n) \ |\Sigma|)$.

As we mentioned in the Introduction, the main drawback of the FM-index is that, hidden in the $o(n)$ term of the space bound, there are constants which depend exponentially on the alphabet size $|\Sigma|$. In Section 4.5 we describe our new implementation of the FM-index concept, which takes $nH_k(T)+o(n)$ bits and allows the computation of $\mathsf{Occ}(c,q)$ and $T^{bwt}[i]$ in $O(1)$ time for a reasonable range of alphabet sizes, i.e. $|\Sigma| = O(\text{polylog}(n))$.

4.4 Compression Boosting

The concept of *compression boosting* has been recently introduced in [6, 10, 13] opening the door to a new approach to data compression. The key idea is that one can take an algorithm whose performance can be bounded in terms of the zero-order entropy and obtain, via the booster, a new compressor whose performance can be bounded in terms of the k-th order entropy, *simultaneously for all k*. Putting it another way, one can take a compression algorithm that uses no context information at all and, via the boosting process, obtain an algorithm that automatically uses the "best possible" contexts.

For technical reasons we need a boosting theorem which is slightly different from the one in [6, 10]. However, the proof of Theorem 4 is obtained by a straightforward modification of the proof of Lemma 3.5 in [10].

Theorem 4. *Let $\mathcal{A}$ be an algorithm which compresses any string S in less than $|S|H_0(S) + f(|S|)$ bits, where $f(\ )$ is a non decreasing concave function. Given $T[1,n]$ there is an $O(n)$ time procedure that computes a partition $S_1, S_2, \ldots, S_z$ of T^{bwt} such that, for any $k \geq 0$, we have*

$$\sum_{i=1}^{z} |\mathcal{A}(S_i)| \ \leq \ \sum_{i=1}^{z} (|S_i|H_0(S_i) + f(|S_i|)) \ \leq \ nH_k(T) + |\Sigma|^k f(n/|\Sigma|^k).$$

Proof. For any $k \geq 0$ let $\hat{S}_1, \hat{S}_2, \ldots, \hat{S}_m$ denote the partition of T^{bwt} such that

$$\sum_{i=1}^{m} |\hat{S}_i|H_0(\hat{S}_i) = nH_k(T). \tag{3}$$

Each $\hat{S}_i$ is a permutation of one of the strings w_T defined in Sec. 4.1 with $w \in \Sigma^k$ (see [10, Sec. 2.2] for details). Repeating verbatim the proof of Lemma 3.5 in [10] we get that the partition $S_1, \ldots, S_z$ of T^{bwt} produced by the boosting algorithm is such that

$$\sum_{i=1}^{z} (|S_i|H_0(S_i) + f(|S_i|)) \leq \sum_{i=1}^{m} \left(|\hat{S}_i|H_0(\hat{S}_i) + f(|\hat{S}_i|)\right). \tag{4}$$

Since by hypothesis $|\mathcal{A}(S_i)| \leq |S_i|H_0(S_i) + f(|S_i|)$, From (4) and (3) we get

$$\begin{aligned}\sum_{i=1}^{z} (|S_i|H_0(S_i) + f(|S_i|)) &\leq \sum_{i=1}^{m} |\hat{S}_i|H_0(\hat{S}_i) + \sum_{i=1}^{m} f(|\hat{S}_i|) \\ &= nH_k(T) + \sum_{i=1}^{m} f(|\hat{S}_i|) \\ &\leq nH_k(T) + |\Sigma|^k f(n/|\Sigma|^k),\end{aligned}$$

where the last inequality follows from the concavity of $f(\ )$ and the fact that $m \leq |\Sigma|^k$. □

To understand the relevance of this result suppose that we want to compress $T[1,n]$ and that we wish to exploit the zero-order compressor $\mathcal{A}$. Using the boosting technique we can first compute the partition $S_1, S_2, \ldots, S_z$ of T^{bwt}, and then compress each S_i using $\mathcal{A}$. By the above theorem, the overall space occupancy would be bounded by $\sum_i |\mathcal{A}(S_i)| \leq nH_k(T) + |\Sigma|^k f(n/|\Sigma|^k)$. Note that the process is reversible, because the decompression of each S_i retrieves T^{bwt}, and from T^{bwt} we can retrieve T using the inverse BWT. Summing up, the booster allows us to compress T up to its k-th order entropy using only the zero-order compressor $\mathcal{A}$. Note that the parameter k is neither known to $\mathcal{A}$ nor to the booster, it comes into play only in the space complexity analysis. This means that the space bound in Theorem 4 holds *simultaneously* for all $k \geq 0$. The only information that is required by the booster is the function $f()$ such that $|S|H_0(S) + f(|S|)$ is an upper bound on the size of the output produced by $\mathcal{A}$ on input S.

4.5 An Improved FM-Index Implementation

We now show how to combine the tools described in the previous sections to obtain an FM-index implementation with query time independent of the alphabet size when $|\Sigma| = O(\text{polylog}(n))$. At the end we consider the case of larger alphabets.

The crucial observation is the following. To build the FM-index we need to solve two problems: (a) compress T^{bwt} up to $H_k(T)$, and (b) support the efficient computation of $\mathsf{Occ}(c,q)$ and $T^{bwt}[q]$. We use the boosting technique to transform problem (a) into the problem of compressing the strings $S_1, S_2, \ldots, S_z$ up to their zero-order entropy, and use the generalized wavelet tree to create a compressed (up to H_0) and indexable representation of each S_i, thus solving simultaneously problems (a) and (b).

1. Use Theorem 4 to determine the optimal partition $S_1, S_2, \ldots, S_z$ of T^{bwt} with respect to $f(t) = Kt/\log^{\varepsilon} t + (1 + |\Sigma|)\log n$, where K is larger than the constant hidden in the $O(t/\log^{\varepsilon} t)$ term of Theorem 2.

2. Build a binary sequence $\mathcal{B}$ that keeps track of the starting positions in T^{bwt} of the S_i's. The entries of $\mathcal{B}$ are all zeroes except for the bits at positions $\sum_{j=1}^{i} |S_j|$ for $i = 1, \ldots, z$ which are set to 1. Construct the data structure of Theorem 1 (or the one in [37]) over the binary string $\mathcal{B}$.

3. For each string S_i, $i = 1, \ldots, z$ build:
   (a) the array $\mathcal{C}_i[1, |\Sigma|]$ such that $\mathcal{C}_i[c]$ stores the occurrences of character c within $S_1 S_2 \cdots S_{i-1}$;
   (b) the generalized wavelet tree $\mathcal{T}_i$ of Theorem 2.

Fig. 5. Construction of our improved FM-index.

The details of the construction are given in Fig. 5, some comments follow. To compute $T^{bwt}[q]$, we first determine the substring S_y containing the q-th character of T^{bwt} by computing $y = \mathsf{Rank}_1(\mathcal{B}, q)$. Then we exploit the generalized wavelet tree $\mathcal{T}_y$ to determine $T^{bwt}[q]$. By Theorem 1 the former step takes $O(1)$ time, and by Theorem 2 the latter step takes also $O(1)$ time.

To compute $\mathsf{Occ}(c,q)$, we initially determine the substring S_y of T^{bwt} where the matrix row q occurs in, $y = \mathsf{Rank}_1(\mathcal{B}, q)$. Then we find the relative position of q within S_y by calculating $q' = q - \sum_{j=1}^{y-1} |S_j| = q - \mathsf{Select}_1(\mathcal{B}, y)$.[2] Finally, we exploit the generalized wavelet tree $\mathcal{T}_y$ and use the array $\mathcal{C}_y[c]$ to compute $\mathsf{Occ}(c,q) = \mathsf{Occ}_{S_y}(c, q') + \mathcal{C}_y[c] = \mathsf{Rank}_c(S_y, q') + \mathcal{C}_y[c]$. Again, by Theorems 1 and 2 this computation takes overall $O(1)$ time. Using this technique inside algorithms count and locate of Section 4.3, we immediately obtain $O(p)$ time to count the occurrences of a

[2] Instead of implementing *select* we can just store all these partial sums at $O(z \log n)$ extra space, where z is the number of pieces in the T^{bwt} partition, and this cost does not affect the overall space result.

pattern $P[1,p]$ and $O(\log^{1+\varepsilon} n)$ time to retrieve the position of each occurrence. The time to display a substring of length ℓ is $O(\ell)$ in addition to the locate time.

We now analyze the space occupancy of our data structure. Let us call a sequence S_i *long* if $|\Sigma_i| = O(\mathrm{polylog}(|S_i|))$, where Σ_i is the alphabet of S_i. Otherwise S_i is *short*.

Let us first assume that all sequences are long. This assumption and Theorem 2 allow us to conclude that a generalized wavelet tree $\mathcal{T}_i$ built on S_i uses $|S_i|H_0(S_i) + O(|S_i|/\log^{\varepsilon}|S_i|)$ bits, for any $0 < \varepsilon < 1$. By Theorem 1, the storage of $\mathcal{B}$ takes $\lceil \log \binom{n}{z} \rceil + O((n \log\log n)/\log n) \le z \log n + O((n \log\log n)/\log n)$ bits. Each array $\mathcal{C}_i$ takes $|\Sigma| \log n$ bits. Consequently, under the hypothesis that $|\Sigma_i| = O(\mathrm{polylog}(|S_i|))$ for all sequences S_i, the total space occupancy is bounded by

$$\sum_{i=1}^{z}\Big(|S_i|H_0(S_i) + K|S_i|/\log^{\varepsilon}|S_i| + (1+|\Sigma|)\log n\Big) + O((n\log\log n)/\log n)\,.$$

Note that the function $f(t)$ used at Step 1 of our construction (see Fig. 5) matches exactly the overhead with respect to H_0 that we have for each S_i. From Theorem 4 we get that for any $k \ge 0$ we can bound the above summation by

$$nH_k(T) + O\Big(n/\log^{\varepsilon}(n/|\Sigma|^k)\Big) + O\Big(|\Sigma|^{k+1}\log n\Big) + O((n\log\log n)/\log n)\,. \qquad (5)$$

Recall that we are interested in bounding the space occupancy in terms of $H_k(T)$ only for $k \le \alpha \log_{|\Sigma|} n$ for some $\alpha < 1$. In this case we have $|\Sigma|^k \le n^{\alpha}$. By observing that $O((n\log\log n)/\log n) = O(n/\log^{\varepsilon} n)$ we turn (5) into

$$nH_k(T) + O(n/\log^{\varepsilon} n)\,. \qquad (6)$$

We complete the analysis of the space occupancy by considering the case of short sequences, that is, where $|\Sigma_i| = \omega(\mathrm{polylog}(|S_i|))$ for some sequences S_i. The first part of Theorem 2 implies that we can always choose $r = |\Sigma_i|^{1/\kappa}$ such that all the query times are the constant $O(\log_r |\Sigma_i|) = O(\kappa)$. The extra space, however, can only be bounded by $o(|\Sigma_i||S_i|)$. Since $\omega(\mathrm{polylog}(|S_i|)) = |\Sigma_i| \le |\Sigma| = O(\mathrm{polylog}(n))$, we must have $|S_i| = o(n^{\beta})$ for any $\beta > 0$. Thus, the extra space for a short sequence S_i is $o(|\Sigma||S_i|) = o(n^{\beta}\mathrm{polylog}(n))$.

Recalling again that we are interested in bounding the space occupancy in terms of $H_k(T)$ only for $k \le \alpha \log_{|\Sigma|} n$ and $\alpha < 1$, we have that the overall space overhead because of the (at most $|\Sigma|^k \le n^{\alpha}$) short sequences S_i is $o(n^{\alpha+\beta}\mathrm{polylog}(n))$ bits for any $\beta > 0$. If we take $\beta < 1 - \alpha$, the space bound becomes $O(n/\log^{\varepsilon} n)$ for any desired $0 < \varepsilon < 1$. Therefore, we can correctly apply Theorem 4 since the extra space to represent short sequences can be considered to be the one corresponding to a long sequence plus smaller terms that are left in the sublinear term that accompanies $nH_k(T)$.

We note that we have inherited from (6) a sublinear space cost of the form $O(n/\log^{\varepsilon} n)$, for any $0 < \varepsilon < 1$. Also, from Theorem 3, we carry another term of the form $O(n/\log^{\varepsilon} n)$, for any $\varepsilon > 0$. We then achieve the following result:

Theorem 5. *Let $T[1,n]$ be a string over an alphabet Σ, where $|\Sigma| = O(\mathrm{polylog}(n))$. The data structure of Fig. 5 indexes $T[1,n]$ within $nH_k(T) + O(n/\log^{\varepsilon} n)$ bits, for any $k \le \alpha \log_{|\Sigma|} n$ and $0 < \alpha, \varepsilon < 1$. It can count the number of occurrences of any string $P[1,p]$ in T in $O(p)$ time, locate each occurrence in $O(\log^{1+\varepsilon} n)$ time, and display any text substring of length ℓ in $O(\ell + \log^{1+\varepsilon} n)$ time.* □

Large Alphabets. Suppose now that the alphabet is larger than $O(\text{polylog}(n))$, in particular $|\Sigma| = O(n^\beta)$ with $\beta < 1$. Corollary 1 shows that we can obtain $O(\log|\Sigma|/\log\log n)$ query time on the sequences S_i, using space $|S_i|H_0(S_i) + O(|\Sigma|\log|S_i|) + o(|S_i|\log|\Sigma|)$ (more precisely, the latter term is $O(|S_i|\log|\Sigma|/\log\log|S_i|)$ in Corollary 1). It is easy to repeat the analysis and obtain that, instead of (5), the bound on the size of our index becomes:

$$nH_k(T) + O\Big(n\log|\Sigma|/\log\log(n/|\Sigma|^k)\Big) + O\Big(|\Sigma|^{k+1}\log n\Big) + O((n\log\log n)/\log n)\,.$$

Since, for any $k \le \alpha(\log_{|\Sigma|} n) - 1$, with $0 < \alpha < 1$, the above bound is $nH_k(T) + o(n\log|\Sigma|)$ we can state the following theorem.

Theorem 6. *Let $T[1,n]$ be a string over an alphabet Σ, with $|\Sigma| = O(n^\beta)$ and $\beta < 1$. It is possible to index $T[1,n]$ within $nH_k(T) + o(n\log|\Sigma|)$ bits, for any $k \le \alpha(\log_{|\Sigma|} n) - 1$ and $0 < \alpha < 1$. This index can count the number of occurrences of any string $P[1,p]$ in T in $O(p\log|\Sigma|/\log\log n)$ time, locate each occurrence in $O(\log|\Sigma|\log^{1+\varepsilon} n/\log\log n)$ time, and display any text substring of length ℓ in $O((\ell + \log^{1+\varepsilon} n)\log|\Sigma|/\log\log n)$ time.* □

As an alternative to Theorem 6, we can handle large alphabets by decreasing the arity r of our generalized wavelet tree. This reduces the space occupancy of our index at the cost of an increased query time. Here we only discuss the extreme case $r = 2$ in which we use the traditional binary wavelet tree instead of our sequence representation. Using binary wavelet trees we can represent each S_i in $|S_i|H_0(S_i) + O(\log|\Sigma|(|S_i|\log\log|S_i|)/\log|S_i|)$ bits of storage. Combining this representation with the compression booster we get an index of size bounded by $nH_k(T) + O(\log|\Sigma|(n\log\log n/\log n))$ bits for any $k \le \alpha\log_{|\Sigma|} n$ and $\alpha < 1$. Notice that this holds for any alphabet size such that $|\Sigma| = o(n/\log n)$. Since querying a binary wavelet tree takes $O(\log|\Sigma|)$ time, our simplified index can count the number of occurrences in $O(p\log|\Sigma|)$ time, locate each occurrence in $O(\log|\Sigma|(\log^2 n/\log\log n))$, and display any text substring of length ℓ in $O(\log|\Sigma|(\ell + \log^2 n)/\log\log n))$ time. The details on the analysis of the above simplified index can be found in [11].

5 Conclusions

The contribution of this paper is twofold. First, we have presented a compressed representation of sequences $S[1,n]$ that requires $nH_0(S) + o(n)$ bits of space and is able to retrieve individual symbols $S[q]$ and answer *rank* and *select* queries in constant time. The technique works whenever the alphabet size of the sequence satisfies the condition $|\Sigma| = O(\text{polylog}(n))$. This is a non trivial generalization of previous results on binary sequences [37] and an improvement of previous results on general sequences [18].

Secondly, we have combined the above result with an existing compression boosting technique to obtain a compressed full-text index for a text $T[1,n]$ which uses $nH_k(T) + o(n)$ bits whenever the alphabet size is $O(\text{polylog}(n))$. Our index has the smallest known space occupancy, is a self-index, is faster than other indexes of the same size, and is the first compressed index with query time independent of the alphabet size. We have also shown that on larger alphabets we can still improve the best existing results.

After the first publication of this article, some new techniques have been proposed that enable improving our bounds on *rank* and *select* queries slightly: Golynksi, Munro, and Rao [14] propose a succinct structure that supports *rank* in $O(\log\log\sigma)$ time and *select* in constant time. Their structure is, however, not compressed, as it uses $n\log\sigma + o(n\log\sigma)$ bits. Hence, it can not be coupled with the compression boosting technique. Sadakane and Grossi [40] propose a technique to represent a sequence X in $nH_k(X) + o(n\log|\Sigma|)$ bits supporting constant access to its short substrings. However, even using this more powerful representation, compression boosting appears to be still necessary to achieve a space usage bounded by $nH_k(T) + o(n)$ bits (see [16]).

There are several future challenges on compressed full-text indexes: (*i*) obtaining better results when the alphabet size is not $O(\text{polylog}(n))$; (*ii*) removing the limit on the maximum entropy order k that can be achieved, which is currently $k \le \alpha\log_{|\Sigma|} n$ for any $0 < \alpha < 1$; (*iii*) achieving the optimal query times within $nH_k(T) + o(n)$ space, that is, $O(p\,/\log_{|\Sigma|} n)$ for counting and $O(1)$ for locating, as opposed to our $O(p)$ and $O(\log^{1+\varepsilon} n)$ (this has been partially achieved [20, 8] in some cases); (*iv*) coping with updates to the text and secondary memory issues (see e.g. [8, 2, 25]); (*v*) handling more sophisticated searching, such as approximate and regular expression searching (see e.g. [23]).

It should be clear, however, that some limits cannot be surpassed. For example, one cannot achieve an $nH_k(T) + o(n)$ space bound without any restriction on $|\Sigma|$ or k. To see this, consider the extreme case in which $|\Sigma| = n$, that is, the input string consists of a permutation of n distinct characters. In this case we have $H_k(T) = 0$ for all $k \ge 1$, and any representation of such string must require $\Omega(n\log n)$ bits. Hence a self-index of size $nH_k(T) + o(n)$ bits cannot exist. Understanding which are the limits and the space-time tradeoffs that can be achieved on compressible strings (cfr. [5]) is an extremely interesting open problem.

References

1. M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
2. H. Chan, W. Hon, and T. Lam. Compressed index for a dynamic collection of texts. In *Proc. Symposium on Combinatorial Pattern Matching (CPM 2004)*, Springer-Verlag, LNCS 3109, pages 445–456, 2004.
3. D. Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, 1996.
4. M. Crochemore and W. Rytter. *Text algorithms.* Oxford University Press, 1994.
5. E. D. Demaine and A. López-Ortiz. A linear lower bound on index size for text retrieval. *Journal of Algorithms*, 48(1):2–15, 2003.
6. P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, 52(4):688–713, 2005.
7. P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc. IEEE Symposium on Foundations of Computer Science (FOCS 2005)* , pages 184–193, 2005.
8. P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005. Earlier in *FOCS 2000.*
9. P. Ferragina and G. Manzini. An experimental study of a compressed index. *Information Sciences: special issue on "Dictionary Based Compression"*, 135:13–28, 2001. Earlier in *SODA 2001.*
10. P. Ferragina and G. Manzini. Compression boosting in optimal linear time using the Burrows-Wheeler transform. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)*, pages 655–663, 2004.
11. 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 2004)*, Springer-Verlag, LNCS 3246, pages 150–160, 2004.
12. R. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. In *Proc. Symposium on Combinatorial Pattern Matching (CPM 2004)*, Springer-Verlag, LNCS 3109, pages 159–172, 2004.
13. R. Giancarlo and M. Sciortino. Optimal partitions of strings: A new class of Burrows-Wheeler compression algorithms. In *Proc. Symposium on Combinatorial Pattern Matching (CPM 2003)*, pages 129–143, 2003.

14. A. Golynski, I. Munro, and S. S. Rao. Rank/Select Operations on Large Alphabets: a Tool for Text Indexing. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA 2006)*, pages 368–373, 2006.
15. G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In B. Frakes and R. A. Baeza-Yates and, editors, *Information Retrieval: Data Structures and Algorithms*, chapter 5, Prentice-Hall, pages 66–82, 1992.
16. R. González and G. Navarro. Statistical Encoding of Succinct Data Structures. Technical Report TR/DCC-2006-1, Dept. of Computer Science, University of Chile, January 2006.
17. Sz. Grabowski, V. Mäkinen, G. Navarro, and A. Salinger. A Simple Alphabet-Independent FM-Index. In *Proc. 10th Prague Stringology Conference (PSC 2005)*, pages 230–244, 2005. Earlier short abstract in *SPIRE 2004*.
18. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*, pages 841–850, 2003.
19. R. Grossi, A. Gupta, and J. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)*, pages 636–645, 2004. Extended version to appear in *ACM Transactions on Algorithms*.
20. R. Grossi and J. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM Journal on Computing*, 35(2):378–407, 2005. Earlier in *STOC 2000*.
21. J. Healy, E.E. Thomas, J.T. Schwartz, and M. Wigler. Annotating large genomes with exact word matches. *Genome Research*, 13:2306–2315, 2003.
22. W.-K. Hon, T.-W. Lam, W. K. Sung, W. L. Tse, C.-K. Wong, and S. M. Yiu. Practical Aspects of Compressed Suffix Arrays and FM-Index in Searching DNA Sequences. In *Proc. Workshop on Algorithm Engineering and Experiments (ALENEX 2004)*, SIAM, pp. 31–38, 2004.
23. N. Huynh, W. Hon, T. Lam and W. Sung. Approximate string matching using compressed suffix arrays. In *Proc. Symposium on Combinatorial Pattern Matching (CPM 2004)*, Springer-Verlag, LNCS 3109, pages 434–444, 2004.
24. G. Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie Mellon University, 1989.
25. W. Hon, T. Lam, K. Sadakane, W. Sung and S. Yiu. Compressed index for dynamic text. In *Proc. IEEE Data Compression Conference (DCC 2004)*, pages 102–111, 2004.
26. V. Mäkinen and G. Navarro. Compressed compact suffix arrays. In *Proc. Symposium on Combinatorial Pattern Matching (CPM'04)*, Springer-Verlag, LNCS 3109, pages 420–433, 2004.
27. V. Mäkinen and G. Navarro. New search algorithms and time/space tradeoffs for succinct suffix arrays. Technical Report C-2004-20, University of Helsinki, Finland, 2004.
28. V. Mäkinen and G. Navarro. Succinct Suffix Arrays based on Run-Length Encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005. Earlier in *CPM 2005*.
29. V. Mäkinen, G. Navarro, and K. Sadakane. Advantages of backward searching — efficient secondary memory and distributed implementation of compressed suffix arrays. In *Proc. International Symposium on Algorithms and Computation (ISAAC 2004)*, Springer-Verlag, LNCS 3341, pages 681–692, 2004.
30. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
31. G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
32. I. Munro. Tables. In *Proc. Foundations of Software Technology and Theoretical Computer Science (FSTTCS 1996)*, Springer-Verlag, LNCS 1180, pages 37–42, 1996.
33. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proc. IEEE Symposium on Foundations of Computer Science (FOCS 1997)*, pages 118–126, 1997.
34. J. Munro, R. Raman, V. Raman, and S. Rao. Succinct representations of permutations. In *Proc. International Colloquium on Automata, Languages and Programming (ICALP 2003)*, Springer-Verlag, LNCS 2719, pages 345–356, 2003.
35. G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms*, 2(1):87–114, 2004.
36. R. Pagh. Low redundancy in dictionaries with $O(1)$ worst case lookup time. In *Proc. International Colloquium on Automata, Languages and Programming (ICALP 1999)*, Springer-Verlag, LNCS 1644, 595–604, 1999.
37. R. Raman, V. Raman, and S.Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*, pages 233–242, 2002.
38. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003. Earlier in *ISAAC 2000*.
39. K. Sadakane. Succinct representations of LCP information and improvements in the compressed suffix arrays. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*, pages 225–232, 2002.
40. K. Sadakane and R. Grossi. Squeezing Succinct Data Structures into Entropy Bounds. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA 2006)*, pages 1230–1239, 2006.

41. K. Sadakane and T. Shibuya. Indexing huge genome sequences for solving various problems. *Genome Informatics*, 12:175–183, 2001.
42. J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Transaction on Information Theory*, 24:530–536, 1978.