Random Access to High-Order Entropy Compressed Text

Roberto Grossi

Dipartimento di Informatica, Università di Pisa, Italy
grossi@di.unipi.it

Abstract. This paper is a survey on the problem of storing a string in compressed format, so that (*a*) the resulting space is close to the high-order empirical entropy of the string, which is a lower bound on the compression achievable with text compressors based on contexts, and (*b*) constant-time access is still provided to the string as if it was uncompressed. This is obviously better than decompressing (a large portion of) the whole string each time a random access to one of its substrings is needed. A storage scheme that satisfies these requirements can thus replace the trivial explicit representation of a text in any data structure that requires random access to it, alleviating the algorithmic designer from the task of compressing it.

1 Introduction

Massive textual data and text with markup are the formats of choice for documents, data interchange, document databases, data backup, log analysis, and configuration files. These forms of unstructured, semi-structured, and replicated data are gaining increasing popularity. Most of their processing is in the main memory in computing clusters where space is an important issue. At the same time, they are often highly compressible when considered as strings.

While most sequential processing methods decompress on the fly and scan the content of these compressed strings (e.g. using tools such as `zcat` or `bzcat`), there are many situations in which fast *random* access to some selected contiguous segments (substrings) of these compressed strings is needed. The decompression of the whole strings is too expensive because the accessed substrings may be relatively few and small, and potentially scattered through the memory.

Problem Statement. The above scenario motivates the introduction of a *compressed string storage scheme* (CSSS), which stores a string $S[1, n]$ from the alphabet $[\sigma] = \{1, \ldots, \sigma\}$ in compressed form, while allowing random access to the string via the operation:

Access(i, m)**:** return the substring $S[i, i+m-1]$ for $m \geq 1$ and $1 \leq i \leq n-m+1$.

The *dynamic* version of a CSSS supports also update operations on S, namely, insertions, deletions or substitutions of symbols in S.

Model and Complexity. We adopt the standard RAM model with a word size of $\Theta(\lg n)$ bits. We also assume that each symbol of the alphabet $[\sigma]$ is encoded by $\lceil \lg \sigma \rceil$ bits[1] This means that the uncompressed string S requires $n\lceil \lg \sigma \rceil$ bits to be stored in *raw* form, and Access to any substring of m symbols can be optimally (and trivially) performed in $\Theta(1 + m \lg \sigma / \lg n)$ time as each word stores $\Theta(\lg n / \lg \sigma)$ symbols.[2]

When S is highly compressible, which is the case in almost all large-scale texts, the compressed representation of S can reach its information theoretic minimum, called entropy. For each $c \in [\sigma]$, let $p_c = n_c/n$ be its empirical probability of occurring in S, where n_c is the number of occurrences of symbol c in S. The *zero-th order empirical entropy* of S is defined as $H_0(S) = -\sum_{c=1}^{\sigma} p_c \lg p_c$, where $H_0(S) \leq \lg \sigma$. Popular compressors such as those based in Huffman coding and Arithmetic coding are able to store S using $nH_0(S)$ bits plus some lower order term. Note that this compressed representation is effective when $H_0(S) < \lg \sigma$. But this is not theoretically the best for a compressible text.

We can exploit the fact that a certain portion of S, say ω, is often followed by a symbol c in S. In other words, the number of occurrence of ω and ωc are very close. The *conditional* probability of finding c after reading ω in S is therefore very high, potentially much larger than p_c. This translates into a better entropy notion. For any string ω of length k, let ω_S be the string of single symbols following the occurrences of ω in S, taken from left to right, and $|\omega_S|$ be the length of ω_S. The *kth order empirical entropy* of S is defined as $H_k(S) = \frac{1}{n} \sum_{\omega \in [\sigma]^k} |\omega_S| \, H_0(\omega_S)$. Not surprisingly, for any $k \geq 0$ we have $H_k(S) \geq H_{k+1}(S)$. The value of $nH_k(S)$ bits is a lower bound to the output size of any compressor that encodes each symbol of S only considering the symbol itself and the k immediately preceding symbols [24].

Requirements on Time and Space Bounds. Based on the above discussion, we require that a CSSS for string S fulfills the following conditions:

- It supports Access optimally in $\Theta(1 + m \lg \sigma / \lg n)$ time for decoding any substring of S of length m.
- It takes $n\,(H_k(S) + \rho(k, \sigma, n))$ bits, where $\rho(k, \sigma, n)$ is the *redundancy* per symbol, or any additional space required to support random access.

The rationale for the above two conditions is that a CSSS for S can replace S itself for any RAM algorithm A having S as input. This replacement does not penalize the asymptotic time complexity of A, and its space complexity is not worsened when $H_k(S) + \rho(k, \sigma, n) \leq \lceil \lg \sigma \rceil$. For highly compressible S, it is actually $H_k(S) + \rho(k, \sigma, n) = o(\lg \sigma)$, thus showing the importance of using a CSSS for massive texts.

The redundancy $\rho(k, \sigma, n)$ is a quantity of significant fundamental interest, particularly for lower bounds (see [28] and references therein), and it is critical

[1] The logarithms are to the base 2 unless otherwise specified, and $\sigma \leq n$ is customarily taken to be a (usually slowly-growing) function of n.

[2] Several authors prefer to use $\lg_\sigma n$ in place of $\lg n / \lg \sigma$.

in practice. We anticipate that the best redundancy is currently $\rho(k, \sigma, n) = O\left(\frac{\lg \sigma}{\lg n}(k \lg \sigma + \lg \lg n)\right)$, which holds *simultaneously* for all $0 \leq k \leq \lg n / \lg \sigma$. As k increases, the H_k term decreases, but $\rho(k, \sigma, n)$ increases. However, as long as $k = o(\lg n / \lg \sigma)$, we have that the term $n\rho(k, \sigma, n) = o(n \lg \sigma)$ is asymptotically smaller than the space required by S in raw form. Interestingly, no non-trivial lower bounds on the redundancy $\rho(k, \sigma, n)$ are known. The rest of the paper describes the state of the art on this topic.

Impact on Succinct Data Structures. CSSS is a natural fit for *systematic* data structures for texts, also known as *succinct indexes* for texts, where the indexing data structure is separated from the input string S. This concept was born for proving lower bounds [7,13,14], and rapidly extended to to analyze the upper bounds [33,1] on the space required to encode some data structures. Non-systematic data structures instead encode both S and the index data structure together, with no clear separation between the two objects. Some of the advantages of systematic data structures are pointed out in [1] and rephrased here in terms of CSSS.

(1) A systematic data structure does not make assumptions on S, which can therefore be replaced by a CSSS for S thus providing high-order entropy bounds, namely, $n\,(H_k(S) + \rho(k, \sigma, n))$ bits plus the space required by the succinct index. Note that a non-systematic data structure requires instead S to be stored in a specific format.

(2) The same CSSS for S can be shared among several systematic data structures. Here, we can build several succinct indexes on the same CSSS for S without introducing replication. Note that multiple non-systematic data structures must instead replicate S (or its equivalent format) internally.

These features can be made effective by showing how to obtain high-order entropy bounds with static systematic data structures on texts.

Lemma 1 (Barbay et al. [1], Sadakane and Grossi [33]). *Given a* CSSS *for a string S of length n over the alphabet $[\sigma]$, let $n\,(H_k(S) + \rho(k, \sigma, n))$ be its number of required bits. Let $I_1, I_2, \ldots, I_d$ be static systematic data structures defined on the same S, where each I_j uses $n\,\tau_j(\sigma, n)$ bits, without accounting for the storage of S. It is possible to build a static succinct data structure G supporting all the functionalities of $I_1, I_2, \ldots, I_d$ in their same asymptotic time costs, namely, if a functionality takes $t(n)$ time in some I_j, it now still takes $O(t(n))$ time in G. The overall number of bits for G is*

$$n\left(H_k(S) + \rho(k, \sigma, n) + \sum_{j=1}^{d} \tau_j(\sigma, n)\right).$$

The global succinct data structure G in Lemma 1 strips the representation of S from each I_j, and uses a shared CSSS for S. When the algorithms for the functionalities of $I_1, I_2, \ldots, I_d$ need access to S, simply G calls Access on the CSSS for S. The functionalities have only a constant slow-down factor in their time complexities, while the space occupancy greatly reduces from representing d times the same S in some (unknown) format versus a single and optimal entropy-encoded representation of S by the CSSS. In many applications it is often the case that $\rho(k, \sigma, n) + \sum_{j=1}^{d} \tau_j(\sigma, n) = o(\lg \sigma)$, making the simple idea behind Lemma 1 very useful.

Paper Organization. Section 2 describes the basic scheme that is shared by the static CSSS presented in Section 3 and the dynamic CSSS presented in Section 4. Finally, some further discussion and conclusions are given in Section 5.

2 Basic Scheme

As widely used in practice, the basic approach employs a compressor C of choice. It partitions S into disjoint substrings called *blocks* $B_1, B_2, \ldots, B_r$, and stores S as the sequence $Z = C(B_1) \cdot C(B_2) \cdots C(B_r)$ obtained by concatenating the output of C on the blocks. To perform $\mathsf{Access}(i, m)$, it has to find the index j of the block B_j of S that contains position i, and decode $C(B_j), C(B_{j+1}), \ldots,$ until it gets the wanted m symbols. For the sake of discussion, suppose that all blocks in S have the same size w, so that finding j is simple arithmetics, namely, $j - 1 = \lfloor (i-1)/w \rfloor$.

When considering the above approach in terms of the requirements on time and space of a CSSS described in Section 1, we run into a trade-off that can work in some practical cases but it surely causes some theoretical drawbacks. One one hand, if w is much larger than m, then Access has no guarantee to take $\Theta(1 + m \lg \sigma / \lg n)$ time. On the other hand, if w is too small, then Z has not guarantee to use $n\,(H_k(S) + \rho(k, \sigma, n))$ bits of space; for instance, some repetitions inside S of the form xx for some substring x, could be split among two or more blocks $B_j, B_{j+1}, \ldots, B_{j+\ell}$ such that the size of $C(B_j) \cdot C(B_{j+1}) \cdots C(B_{j+\ell})$ is significantly larger than the output size of compressing their concatenation $C(B_j \cdot B_{j+1} \cdots B_{j+\ell})$.

Nevertheless this basic scheme is still good if we replace the black-box compressor C with a pool of suitable succinct data structures. Summing up, we need the following ingredients to implement the basic scheme in a better way, where points 1–2 indicate how to partition S into blocks, and points 3–5 indicate how to encode the sequence of blocks of S (see Fig. 1).

1. *Partition into blocks*: Strategy to partition S into blocks $B_1, B_2, \ldots, B_r$, which can have fixed size w or variable sizes.
2. *Identify the target block*: Succinct data structure(s) or rule to find in $O(1)$ time the index j of the block B_j containing position i, as required by the implementation of $\mathsf{Access}(i, m)$.
3. *Encode a block*: Strategy to assign an encoding $E(B_j)$ to each block B_j, for $1 \leq j \leq r$.
4. *Retrieve a block from its encoding*: Succinct data structure(s) to store, and retrieve in $O(1)$ time, each block B_j from its encoding $E(B_j)$, for $1 \leq j \leq r$.

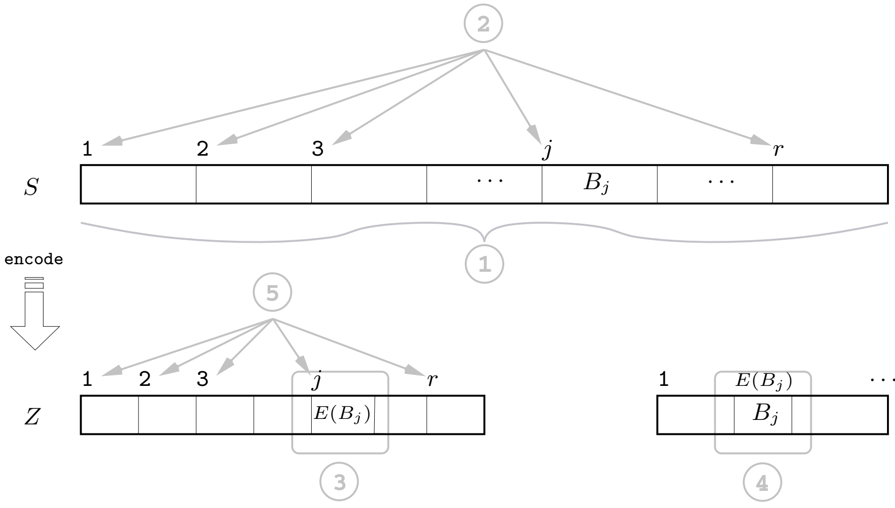

Fig. 1. An illustration of points 1–5 in the basic scheme

5. *Find the encoding of the target block*: Succinct data structure(s) to store $Z = E(B_1) \cdot E(B_2) \cdots E(B_r)$, the concatenation the encodings of the blocks, and access each $E(B_j)$ in $O(1)$ time.

Using the scheme of points 1–5, we can implement $\mathsf{Access}(i, m)$ as follows. We find the index j by points 1–2 and retrieve $E(B_j)$ by point 5. We return B_j by points 3–4. We repeat this for B_{j+1}, B_{j+2}, ..., until we decode the wanted m symbols. As we will see, this implementation requires $O(1)$ time.

In the following we describe, in chronological order, some approaches that follow the above scheme using high-order entropy bounds for the space complexity, employing some basic succinct representation of bitvectors and binary trees that are surveyed in other chapters of this book ([30].). For example, point 3 is easy arithmetics when the blocks have the same size.

When the blocks are of variable sizes, a *fully indexable dictionary* (*FID*) [29] can be employed to mark with a 1 the position in S at the beginning of each block and with 0s the rest of the positions in S. The resulting bitvector X contains r 1s and $n-r$ 0s, and can be stored in the FID using $\lceil \lg \binom{n}{r} \rceil + O(n \lg\lg n / \lg n) = O(r \lg(n/r) + n \lg\lg n / \lg n)$ bits. We remark that this space bound is often negligible when compared to that taken by the other succinct data structures employed in the CSSS.

We recall that a FID supports two basic constant-time operations: $\mathsf{Rank}_b(i)$ returns the number of occurrences of bit $b \in \{0, 1\}$ in the first i positions in X; $\mathsf{Select}_b(i)$ returns the position of the ith occurrence of bit b in X. We can thus locate the block index j, corresponding to the block B_j in S that contains position i as mentioned in point 2, by computing $j = \mathsf{Rank}_1(i)$.

3 Static Schemes

We describe the family of CSSS where only Access operation is supported on input string S, which does not change during its lifetime. The first scheme has variable-size blocks in the partition of S (point 1 in Section 2) but produces fixed-size encodings for them (point 3); the last two schemes have fixed-size blocks for S but produce variable-size encodings for them. Table 1 summarizes some distinguishing features for the three static schemes discussed in this section.

Table 1. Summary of discussed results on static CSSS

redundancy $\rho(k, \sigma, n)$	context size	block size	ref.
$O(\frac{\lg \sigma}{\lg n}((k+1)\lg \sigma + \lg\lg n))$	any k	variable	§3.1
$O(\frac{\lg \sigma}{\lg n}(k \lg \sigma + \lg\lg n))$	fixed $k < (1-\epsilon)\lg n/\lg \sigma$	$(1/2)\lg n/\lg \sigma$	§3.2
$O(\frac{\lg \sigma}{\lg n}(k \lg \sigma + \lg\lg n))$	$k = o(\lg n/\lg \sigma)$	$(1/2)\lg n/\lg \sigma$	§3.3

The three schemes share the same optimal time cost of $\Theta(1 + m \lg \sigma/\lg n)$ for Access, and $O(n \lg \sigma)$ construction time. Space is $n\,(H_k(S) + \rho(k, \sigma, n))$ bits, where the redundancy $\rho(k, \sigma, n)$ per symbol is reported in the table and is slightly higher (a term $(k + 1)$ instead of k) for the first scheme. The first scheme is oblivious with respect to the value of k (which is not part of the input but appears only in the analysis) and allows for potentially long blocks, even though the range of values for $k = \Omega(\lg n/\lg \sigma)$ gives a too large redundancy $\rho(k, \sigma, n) = \Omega(\lg \sigma)$. The second method requires a specific choice of $k < (1 - \epsilon)\lg n/\lg \sigma$, for any $0 < \epsilon < 1$, and all blocks must contain $(1/2)\lg n/\lg \sigma$ symbols. Interestingly, it also supports append operations to add symbols at the end of S in constant amortized time per symbol. The third scheme works for all $k = o(\lg n/\lg \sigma)$ simultaneously, and all blocks must contain $(1/2)\lg n/\lg \sigma$ symbols.

3.1 LZ78 Parsing and Encoding

Sadakane and Grossi [33] introduced the notion here called CSSS, meeting the time and space requirements described in Section 1 with redundancy $\rho(k, \sigma, n) = O(\frac{\lg \sigma}{\lg n}((k + 1)\lg \sigma + \lg\lg n))$ simultaneously for any k.[3] We give a simplified description of the ideas in [34], following the basic scheme of Section 2.

As for point 1, the partitioning of the input string S produces r blocks of variable sizes using first the Ziv-Lempel compression algorithm [38], also known as LZ78 parsing, and then a greedy post-processing.

The LZ78 parsing works as follows. First we initialize a trie T as empty, the current position $p = 1$ in S. Then, we parse S into blocks from left to right,

[3] The authors of [15] pointed out a mistake in the smaller redundancy originally reported in [34] that we fix here in Lemma 2.

finding the longest string $t \in T$ that appears as a prefix of $S[p, n]$ (where t is the empty string when T is empty). Thus we obtain the block $S[p, p+|t|] \equiv t \cdot S[p+|t|]$ to be inserted into T. We set $p = p+|t|+1$, and repeat the parsing to discover the next block. The resulting trie T is called an *LZ-trie*, and r' is the final number of blocks generated by algorithm LZ78 (and thus the number of nodes in T). We use Lemma 2.3 from [20] for bounding r' in terms of the kth-order empirical entropy $H_k(S)$ of the string S.

Lemma 2 (Kosaraju and Manzini [20]). *Let r' be the number of blocks produced by any parsing of the string S, such that each block appears at most M times. For any $k > 0$,*

$$r' \lg r' \leq nH_k(S) + r' \lg \frac{n}{r'} + r' \lg M + \Theta(kr' \lg \sigma)$$

The *greedy post-processing* of the LZ78 parsing with window size w works as follows, for a parameter[4] $w = \frac{1}{2} \lg n / \lg \sigma$. We define a block *short* if it contains less than w symbols, and *long* otherwise. We perform a left-to-right scan of the r' blocks found by the LZ78 parsing, tagging each block as either short or long. However, during this scan, we cluster together maximal runs of consecutive *short* blocks, so that the resulting substring, called *dense block*, is not longer than w symbols: a dense block replaces the short blocks that it contains. Moreover, we impose that any two consecutive dense regions are always separated by a (short or long) block.

This greedy post-processing thus partitions S into r blocks, where $r \leq r'$, satisfying the following conditions.

- The blocks are pairwise disjoint and tagged as either long, short, or dense: by construction, no two consecutive short blocks or dense blocks can exist.
- Any two consecutive blocks contains more than w symbols in total.

As a result, any substring of S of length m overlaps with $O(1 + m/w) = O(1 + m \lg \sigma / \lg n)$ blocks. Retrieving each such block in constant time provides the claimed bound for Access. We thus discuss how to encode the sequence of blocks so that each of them can be retrieved in constant time (see points 3–5 in the basic scheme of Section 2, as we use a FID for point 2 as already discussed).

We consider the r blocks as a set of r strings, which are stored in a fast compressed trie F. Note that we do not specify the details on how to store F since there are many ways described in the chapter of this book dealing with succinct trees [30]. Conceptually F is a refinement of the LZ-trie produced during the LZ78 parsing, augmented with the dense blocks. However, since the dense blocks are of length at most w, there cannot be more than $\sigma^w = O(\sqrt{n})$ distinct ones, so they increment the size of the LZ-trie by $o(n)$ bits when obtaining F.

As a result, for each block B_j of the partition of S we get a unique identifier in $[r]$ using F and vice versa. This identifier is the encoding $E[B_j]$. The theoretical implementation of F so that given $E[B_j]$, we can retrieve B_j in constant time

[4] In practical situations it is more convenient to fix a larger value of w.

is quite complex (see [34]) but practical non-constant implementations can be adapted to this goal using string dictionaries (e.g. [4,27]).

Using the above identifiers, each block B_j is encoded by the $b = \lceil \lg r \rceil$ bits of $E(B_j)$, and the sequence $Z = E(B_1) \cdot E(B_2) \cdots E(B_r)$ is the encoding of the sequence of blocks.[5] In order to retrieve the jth block B_j, we access the jth b-bit integer $E(B_j)$ in Z by simple arithmetics, and use this integer as the identifier for the wanted block. This is given to F as an input query, and the outcome is B_j as explained above.

When computing the space bound, we need $O(r \lg(n/r) + n \lg\lg n/ \lg n)$ bits for the FID in point 2, $O(n \lg\sigma \lg\lg n/\lg n)$ bits for storing F in a succinct way (point 4) given the choice of w, and $r\lfloor \lg r \rfloor \leq r \lg r + r$ bits for Z (point 5). Observing that $r \leq r'$, we can apply Lemma 2 on r, and using the fact that $r \leq 2n/w$ and $M \leq \sigma$ by construction, we can thus bound the space of Z as

$$r \lg r + r \leq nH_k(S) + \frac{2n}{w}(1 + \lg w) + \frac{n}{w} \lg\sigma + \Theta\left(\frac{nk \lg\sigma}{w}\right)$$

The total redundancy $\rho(k, \sigma, n)$ is therefore

$$O\left(\frac{\lg\lg n}{\lg n/\lg\sigma} + \frac{1}{w}\lg w + \frac{1}{w}\lg\sigma + \frac{k\lg\sigma}{w}\right) = O\left(\frac{\lg\sigma}{\lg n}((k+1)\lg\sigma + \lg\lg n)\right)$$

using the choice $w = \frac{1}{2}\lg n/\lg\sigma$.

Theorem 1 (Sadakane and Grossi [34]). *A* CSSS *using LZ78 parsing and encoding can be implemented with redundancy of* $\rho(k, \sigma, n) = O(\frac{\lg\sigma}{\lg n}((k+1)\lg\sigma + \lg\lg n))$ *simultaneously for any* k.

Note that this CSSS actually works with *any* parsing (not only LZ78) that guarantees a high-order entropy bound as stated in Lemma 2. The choice of LZ78 is motivated by the property that the space requirement of the LZ-trie dictionary (and so of the dictionary F) can be shown to be a lower-order term.

3.2 Statistical Encoding

González and Navarro [15] observed that an alternative and simpler CSSS can be obtained by using a semi-static kth-order modeling plus statistical encoding, yielding a redundancy of $\rho(k, \sigma, n) = O(\frac{\lg\sigma}{\lg n}(k \lg\sigma + \lg\lg n))$ for any fixed $k < (1-\epsilon)\lg n/\lg\sigma$ and any constant $0 < \epsilon < 1$.

A semi-static kth-order modeler applied to a string S produces the empirical conditional probability of finding a symbol c after reading substring ω in S (see Model and complexity in Section 1). Formally, $q_1, q_2, \ldots, q_n$ are the empirical probabilities such that $q_i = n_S^c/|\omega_S|$ for $k+1 \leq i \leq n$, where ω_S is the string of single symbols following all the occurrences of substring $\omega \equiv S[i-k, i-1]$ in S,

[5] A smarter encoding from [8] can be employed but the final redundancy does not change asymptotically.

and n_S^c is the number of occurrence of symbol $c \equiv S[i]$ in ω_s. As a result, it is noted in [15] that $\sum_{i=k+1}^{n} -q_i \lg q_i = nH_k(S)$. Hence, any statistical encoder E, such as Arithmetic coding [37], that encodes the ith symbol of S with $-q_i \lg q_i$ bits, has output size $|E(S)| = nH_k(S) + O(k \lg n)$ bits where the additive term is an upper bound for the first k symbols of S.

The above considerations are exploited in [15], as described below by following our basic scheme of Section 2. The input string S is partitioned in blocks of fixed size $w = \frac{1}{2} \lg n / \lg \sigma$. Thus there are $r = \lceil n/w \rceil$ blocks and any block can be located in constant time by simple arithmetics (points 1–2). We therefore discuss how to encode the sequence of blocks so that each of them can be retrieved in constant time (points 3–5).

As for point 3, the semi-static statistical encoder E is applied to the individual blocks using the statistics of their immediately previous k symbols. Namely, for any given block B_j of S, $j > 1$, let κ_j denote the kth-order context of B_j, defined as the last k symbols of B_{j-1} preceding B_j in S. Then, the encoding $E(B_j)$ is obtained using the conditional probabilities of the semi-static kth-order modeler applied to the concatenated sequence $\kappa_j \cdot B_j$. It is worth noting that only the symbols of B_j are encoded, and decoding $E(B_j)$ needs the knowledge of its context κ_j.

As for point 4, we store the mapping between each block B_j and its encoding $E(B_j)$, for $1 \le j \le r$, using a two-dimensional table T of w-long strings, such that $T[\kappa_j, E(B_j)] = B_j$, for $1 \le j \le r$. (Here, κ_1 is the empty string.)

It remains to describe the succinct data structures for storing and accessing the sequence Z of the statistical encodings of the blocks (point 5).

- Define $Z_j = E(B_j)$ if the number of bits $|E(B_j)| \le (1/2) \lg n$, or $Z_j = B_j$ otherwise, for $1 \le j \le r$: store $Z = Z_1 \cdot Z_2 \cdots Z_r$ along with a two-level index [25] to record the lengths $|E(B_j)|$ and mark the starting position of each Z_j inside Z.
- Store a FID D such that $D[j] = \mathtt{1}$ if and only if $Z_j = B_j$, for $1 \le j \le r$. This marks the blocks B_j in S that are stored verbatim because their statistical encodings $E(B_j)$ are too large.
- Store the concatenation of contexts $K = \kappa_1 \cdot \kappa_2 \cdots \kappa_r$ as a long string.

By construction, any substring of S of length m overlaps with $O(1 + m/w) = O(1 + m \lg \sigma / \lg n)$ blocks. Retrieving each such block B_j in constant time provides the claimed bound for Access. To this end, we retrieve Z_j from Z using its two-level index. We check whether $D[j] = \mathtt{1}$ and, if so, we merely return Z_j as $Z_j = B_j$. Otherwise, $Z_j = E(B_j)$: we extract κ_j from K by simple arithmetics, and return $T[\kappa_j, E(B_j)]$ as B_j.

The space requirement of this CSSS can be computed as follows. The two-dimensional table T has size upper bounded by $\sigma^k \times 2^{(1/2) \lg n} \times (1/2) \lg n = O(\sigma^k \sqrt{n} \lg n)$, which is $O(n^{1-\epsilon})$ when $k < (1/2 - \epsilon) \lg n / \lg \sigma$. Playing with the multiplicative constant in the choice of w, we can allow for $k < (1 - \epsilon) \lg n / \lg \sigma$. The storage of D requires $O(r) = O(n \lg \sigma / \lg n)$ bits and that of K takes $O(nk \lg^2 \sigma / \lg n)$ bits. Finally, using Arithmetic coding as encoder E, the storage of Z takes $nH_k(S) + O(k \lg n + r)$ bits, plus $O(n \lg \sigma \lg \lg n / \lg n)$ bits for its two-level index.

Theorem 2 (González and Navarro [15]). *A* CSSS *using semi-static kth-order modeling plus statistical encoding can be implemented with redundancy of* $\rho(k, \sigma, n) = O(\frac{\lg \sigma}{\lg n}(k \lg \sigma + \lg \lg n))$ *for any fixed* $k < (1 - \epsilon) \lg n / \lg \sigma$ *and any constant* $0 < \epsilon < 1$.

Interestingly, this CSSS can also support append operations, where the input string S is extended as $S \cdot c_1 \cdots c_2 \cdots c_g$ by appending symbols $c_1, c_2, \ldots, c_g$. Using the logarithmic method and the global rebuilding technique to dynamize static data structures, the amortized cost is $O(1)$ per appended symbol.

3.3 Frequency Encoding

Ferragina and Venturini [9] coined the term CSSS and described a simplification that avoids the use of LZ-based or statistical compressors, still guaranteeing a redundancy of $\rho(k, \sigma, n) = O(\frac{\lg \sigma}{\lg n}(k \lg \sigma + \lg \lg n))$ simultaneously for all $k = o(\lg n / \lg \sigma)$.

We follow our basic scheme of Section 2 to describe this approach. As done in Section 3.2, the input string S is partitioned into blocks of fixed size $w = \frac{1}{2} \lg n / \lg \sigma$. Thus there are $r = \lceil n/w \rceil$ blocks and each block can be located in constant time by simple arithmetics (points 1–2).

The main idea is how to encode the blocks (points 3–5). We observe that while there are $r = n/w$ blocks, the number of distinct blocks is at most $\sigma^w = \sqrt{n}$. These distinct blocks are sorted in non-increasing order of frequency, namely, according to the number of times each distinct block appears in the partition of S (hence, the total sum of frequencies is r). The distinct blocks thus sorted are then assigned codewords in lexicographic order, starting form the empty string, ϵ, `0`, `1`, `00`, `01`, `10`, `11`, `000`, `001`, `010`, ..., so that less frequent blocks cannot get assigned shorter codewords than more frequent blocks. This is a crucial fact, as we will see. For each block B_j of S, then its encoding $E(B_j)$ is simply the codeword assigned to B_j (as a distinct block) in this way. This completes point 3.

As for point 4, we store the mapping between each block B_j and its encoding $E(B_j)$, for $1 \le j \le r$, using a lookup table T such that $T[E(B_j)] = B_j$, for $1 \le j \le r$.

Finally, we store $Z = E(B_1) \cdot E(B_2) \cdots E(B_r)$, the concatenation the encodings of the blocks, along with a two-level index [25] to mark the starting position of each $E(B_j)$ inside Z (point 5).

By construction, any substring of S of length m overlaps with $O(1 + m/w) = O(1 + m \lg \sigma / \lg n)$ blocks. Retrieving each such block B_j in constant time provides the claimed time bound for Access: retrieve $E(B_j)$ from Z using its two-level index and return $T[E(B_j)]$ as B_j.

The space requirement of this CSSS can be computed as follows. Table T requires $O(\sqrt{n} \lg n)$ bits. The two-level index for Z requires $O(\frac{n \lg \sigma \lg \lg n}{\lg n})$ bits. It is interesting to analyze the space requirement for Z as argued next. Let S_w

be the sequence of blocks as they appear in S. A relevant property is that the 0th-order entropy encoding of S_w gives the kth-order entropy encoding of S.

Lemma 3 (Ferragina and Venturini [9]). *For any $1 \leq w \leq n$, it holds $rH_0(S_w) \leq nH_k(S) + O(rk \lg \sigma)$, simultaneously over all $k \leq w$.*

The codewords $E(B_j)$ assigned to the blocks in S_w attain the 0th-order entropy according to the *golden rule* of data compression: assign shorter codewords to more frequent symbols. The crucial property is therefore that the $\sum_{j=1}^{r} |E(B_j)|$ bits in Z cannot be larger than the 0th-order encoding of S_w.

Theorem 3 (Ferragina and Venturini [9]). *A* CSSS *using frequency encoding can be implemented with redundancy of $\rho(k, \sigma, n) = O(\frac{\lg \sigma}{\lg n}(k \lg \sigma + \lg \lg n))$ simultaneously for all $k = o(\lg n / \lg \sigma)$.*

Fredriksson and Nikitin [12] employed a similar idea of frequency encoding using other forms of codewords but their redundancy is $\rho(k, \sigma, n) = 1 + o(H_k(S) + 1)$, which can be larger than $O(\frac{n \lg \sigma}{\lg n}(k \lg \sigma + \lg \lg n))$. Among others, they adopted the Fibonacci codes [10]: any positive integer x has a unique representation as a sum Fibonacci numbers, such that no two of them are consecutive. Thus the code for x uses $\lg_\phi n + 1$ bits, where ϕ is the golden ratio: since the bit in the last position is `1`, another `1` can be appended so that this is the only position where two consecutive `1`s appear. This fact is useful to locate each $E(B_j)$ inside Z by looking at their unique pattern of consecutive pair of `1`s.

4 Dynamic Schemes

The dynamic version of CSSS is responsive to the changes of the input string S without recomputing from scratch the entire scheme after each update to S. The goal is to guarantee optimal time for Access and still high-order entropy bounds for the space in this dynamic setting. Jansson et al. [19] define the following two variants of dynamic CSSS.

The *compressed random access memory* (*CRAM*) supports Access and

Replace(i, c): replace $S[i]$ by a symbol $c \in [\sigma]$.

The *extended compressed random access memory* (*ECRAM*) supports Access and

Insert(i, c): insert the symbol c into S between positions $i - 1$ (if it exists) and i, and make S one symbol longer.
Delete(i): delete $S[i]$ and make S one symbol shorter.

Table 2 reports the bounds achieved by the known dynamic schemes for CSSS. The space bounds are $n\,(H_k(S) + \rho(k, \sigma, n))$ bits and, for brevity, we use the notation ρ_{stat} in the table as a shorthand for the redundancy bound seen in Section 3.3 for the the static case, namely, $O(\frac{\lg \sigma}{\lg n}(k \lg \sigma + \lg \lg n))$ simultaneously

Table 2. Summary of discussed results on dynamic CSSS

Access(i, m)	Replace	redundancy $\rho(k, \sigma, n)$	ref.
$\Theta(1 + m \lg \sigma / \lg n)$	$O\left(\min\left\{\frac{\lg n}{\lg \sigma}, \frac{(k+1)\lg n}{\lg \lg n}\right\}\right)$	$O(\rho_{\text{stat}})$	§4.1
$\Theta(1 + m \lg \sigma / \lg n)$	$O(1/\epsilon)$	$O(\epsilon(k+1) \lg \sigma + \rho_{\text{stat}})$	§4.1
$\Theta(1 + m \lg \sigma / \lg n)$	$O(1)$	$O(\rho_{\text{stat}})$	§4.2

Access(i, m)	Insert/Delete	redundancy $\rho(k, \sigma, n)$	ref.
$\Theta(\lg n / \lg \lg n + m \lg \sigma / \lg n)$	$\Theta(\lg n / \lg \lg n)$	$O(k \lg \lg n / \lg n + \rho_{\text{stat}})$	§4.1
$\Theta(\lg n / \lg \lg n + m \lg \sigma / \lg n)$	$\Theta(\lg n / \lg \lg n)$	$O(\lg \sigma \lg \lg n / \lg n)$	§4.2

for all $k = o(\lg n / \lg \sigma)$. The reason is that these dynamic schemes use the frequency coding framework presented in Section 3.3 as a baseline.

The top part of Table 2 provides the bounds for the CRAM, and it shows that the CRAM can be implemented with the same optimal Access time and high-order entropy bound of the static CSSS seen in Section 3.

The bottom part of Table 2 provides the bounds for the extended CRAM, noting that Replace can be theoretically simulated by Delete followed by Insert. It is observed in [19] that the Access and Insert/Delete bounds are optimal as the extended CRAM solves the *list representation* problem, for which there is a cell probe lower bound of $\Omega(\lg n / \lg \lg n)$ [11]. There is no dependency on ρ_{stat} and k in the redundancy in the last line because Access has a larger complexity than that in the static case. Finally, note that the scheme in Section 3.2 supports append operations in constant amortized time.

4.1 Managing CRAM and Extended CRAM

Jansson et al. [19] started out from the frequency coding of Ferragina and Venturini [9] (Section 3.3) as a baseline to design a dynamic version of their CSSS. We review the basic scheme of Section 2 to highlight the difficulties of this dynamization process.

First of all, the partition of the input string S into fixed-size blocks (points 1–2) can be handled with some standard techniques for dynamic data structures. Since the block size is $w = (1/2) \lg n / \lg \sigma$, when n grows or shrinks by a constant factor, we rebuilt all the storage scheme using an updated value of w. This has to be incrementally deamortized to provide the worst-case bounds discussed before.

The real difficulty comes with the frequency coding of blocks (points 3–5). The golden rule behind frequency coding is that shorter codewords are assigned to more frequent blocks. When a single symbol is changed in a block, the frequency of the old block should be decreased by one and that of the new block should be either initialized to one (if this is the first time that the block appears) or increased by one. However, it is inefficient to update the lookup table T and recode the entire Z described in Section 3.3, as it could take nearly $O(r)$ time.

The lifetime of the CSSS is therefore divided into phases, where at the end of each phase there is a reconstruction (and deamortization applies too). Two fundamental ideas are employed during a phase.

The first idea is to take a snapshot F_0 of the frequencies of blocks at the beginning of the phase, and maintain an updated version F_1 of the frequencies during the phase. However, F_0 is the one employed to encode the blocks during the phase, while F_1 becomes the new F_0 only in the next phase. This approach makes sense because the high-order empirical entropy of a string does not change too much after a small change to the string.

Lemma 4 (Jansson et al. [19]). *For any two strings T and T' that differ in a single symbol from $[\sigma]$, let $t = \max\{|T|, |T'|\}$. It holds $t\,|H_k(T) - H_k(T')| = O((k+1)(\lg t + \lg \sigma))$.*

We can therefore use F_0 and delay the update of the encodings of the blocks, changing only part of the data structures. However, after some updates, we have to face the costly process of re-encoding. This requires changing Z heavily, which is expensive for several reasons. One of them is that the encoding of the blocks is of variable size, and so we cannot simply maintain Z as the concatenation of the encodings.

Consequently, the second idea is to use a memory-manager data structure to store a set of r variable-length codewords that represent the content of Z. The codewords can change length, up to $\lg r$ bits, while r cannot change during the phase. We want to access the ith codeword and reallocate it when it changes, in constant time, while keeping the wasted space small.

Lemma 5 (Jansson et al. [19]). *Let z be the total number of bits in the encoding of Z. Its r codewords can be stored using a memory manager that occupies $z + O(\lg^4 z + r \lg\lg z)$ bits while supporting the access and the reallocation operations in constant time each.*

Other data structures and invariants are described in [19] to support Access and Replace, while more sophisticated solutions are needed to support Insert and Delete. The CRAM has also a practical implementation [32].

Theorem 4 (Jansson et al. [19]). *The CRAM can be implemented so that* Access *takes optimal $\Theta(1 + m \lg \sigma / \lg n)$ time and* Replace *takes $O(1/\epsilon)$ time for any $\epsilon > 0$, with a redundancy of $\rho(k, \sigma, n) = O(\epsilon(k+1)\lg \sigma + \frac{\lg \sigma}{\lg n}(k \lg \sigma + \lg\lg n))$ simultaneously for all $k = o(\lg n / \lg \sigma)$. The redundancy can be reduced to $O(\frac{\lg \sigma}{\lg n}(k \lg \sigma + \lg\lg n))$ by increasing the cost of* Replace *to $O\left(\min\left\{\frac{\lg n}{\lg \sigma}, \frac{(k+1)\lg n}{\lg\lg n}\right\}\right)$ time.*

Theorem 5 (Jansson et al. [19]). *The extended CRAM can be implemented so that* Access *takes optimal $\Theta(\lg n / \lg\lg n + m \lg \sigma / \lg n)$ time, and* Insert *and* Delete *take $\Theta(\lg n / \lg\lg n)$ time, with a redundancy of $\rho(k, \sigma, n) = O(k \lg\lg n / \lg n + \frac{\lg \sigma}{\lg n}$ $(k \lg \sigma + \lg\lg n))$ simultaneously for all $k = o(\lg n / \lg \sigma)$.*

4.2 Multiple Encodings of Blocks

Grossi et al. [17] conceptually represented S as a sequence S_w of r macro-symbols over the macro-alphabet $[\sigma^w]$, where each macro-symbol is a block of S. They exploited the property in Lemma 3 stating that the 0th-order entropy encoding of S_w gives the kth-order entropy encoding of S.

This implies that if we can maintain a dynamic compressed representation of S_w in $rH_0(S_w)+O(r \lg\lg n)$ bits, we obtain a dynamic compressed representation of S in $nH_k(S) + O(n\frac{\lg \sigma}{\lg n}(k \lg \sigma + \lg\lg n))$ bits as $r = n/w$. Hence, the plan for the frequency coding of blocks (points 3–5) is to use Lemma 5 with $z = rH_0(S_w) + O(r)$ to store the codewords in Z.

We can therefore focus on dynamically maintaining the encoding of the macro-symbols in $[\sigma^w]$ to obtain a 0th-order entropy encoding of S_w. We divide the whole set of assigned codewords into $O(\lg r)$ classes C_j. In the ideal static situation, each macro-symbol y of frequency f_y is assigned a codeword from the class C_j such that $\frac{r}{2^j} < f_y \leq \frac{r}{2^{j+1}}$. Recalling that $\sum_{y \in [\sigma^w]} f_y = r$, the 0th-order entropy plus a lower-order term is achieved for S_w when the codeword for y is $\lg(r/f_y) = j + O(1)$ bits long. This implies that $|C_j| \leq 2^{j+O(1)}$. In the dynamic setting, we need flexibility and so we assign more than one class to y, under the requirement that y has at most one codeword assigned from each such class.

When a symbol of S is replaced, the frequency of a macro-symbol can change. Thus, macro-symbols may move to different classes in the lifetime of the data structure. Once a macro-symbol enters a class for the first time, it is assigned an available codeword e of that class; the next time it will re-enter that class, it will reuse the same codeword e. Since the number of available codewords in any class is limited, it may happen that the last available codeword is consumed in this way (i.e., $|C_j| = 2^{j+O(1)}$). If so, it is shown in [17] that $\Omega(r)$ Replaces have been done, and so we can amortized the cost (which can be deamortized with an incremental rebuilding technique).

This mechanism causes no significant waste of bits in the 0th-order empirical entropy of S_w, since the extra space is a lower-order term: the waste due to multiple codewords (from distinct classes) for the same y is just $O(f_y)$ bits.

Lemma 6 (Grossi et al. [17]). *For any macro-symbol $y \in [\sigma^w]$, the overall space required by the codewords of y in the encoding of S_w is $f_y \lg \frac{r}{f_y} + O(f_y)$ bits.*

The implication of Lemma 6 is that the encoding of S_w takes $\sum_{y \in [\sigma^w]} f_y \lg \frac{r}{f_y} + O(f_y) = rH_0(S_w) + O(r)$ bits.

Other properties and data structures are employed to obtain the bounds for the CRAM and the extended CRAM. In the latter, a variable-size partition of S is maintained.

Theorem 6 (Grossi et al. [17]). *The CRAM can be implemented so that* Access *takes $\Theta(1 + m \lg \sigma / \lg n)$ time and* Replace *takes $O(1)$ time, with a redundancy of $\rho(k, \sigma, n) = O(\frac{\lg \sigma}{\lg n}(k \lg \sigma + \lg\lg n))$ simultaneously for all $k = o(\lg n / \lg \sigma)$.*

Theorem 7 (Grossi et al. [17]). *The extended CRAM can be implemented so that* Access *takes* $\Theta(\lg n/\lg\lg n + m\lg\sigma/\lg n)$ *time, and* Insert *and* Delete *take* $\Theta(\lg n/\lg\lg n)$ *time, with a redundancy of* $\rho(k,\sigma,n) = O(\lg\sigma\lg\lg n/\lg n)$*.*

5 Further Discussion and Conclusions

This paper described a survey on compressed string storage schemes (CSSS) that have optimal access time, as if the text was uncompressed, and squeeze the text of n symbols over alphabet $[\sigma]$ to reach a space bound in bits that is close to the kth-order empirical entropy $nH_k + o(n\lg\sigma)$ (see Table 1). Time bounds for replacing, inserting and deleting individual text symbols are also optimal (see Table 2). Interestingly, there are currently no lower bounds on the redundancy $o(n\lg\alpha)$ when a CSSS has optimal access time. From the practical point of view, some of the proposed methods have been implemented but still the research in this direction has not been fully explored all the directions. For example, one main limitation in practice is that the required value of the block size $w = (1/2)\lg n/\lg\sigma$ is quite small, even for ASCII text. New results in this direction could bring also fresh theoretical questions to investigate.

When removing some of the optimality constraints on a CSSS, such as the constant-time bound for accessing $O(w)$ symbols of the text or the high-order entropy H_k, there are a plethora of ideas and solutions. Indeed, random access to compressed data is a basic problem in many applications on massive data sets, and there are too many practical and effective solutions to be mentioned in this paper (e.g. see the book [36]).

Compressed text indexing is a good source of ideas and sophisticated tools (e.g. see the surveys [16,18,26]) that can achieve high-order H_k entropy bounds. For example, several solutions are based on storing the Burrows-Wheeler transform [5] in some 0th-order compressed data structures, but recovering a substring of the text is suboptimal when compared to the optimal access cost of CSSS. On the other hand, these solutions have indexing and searching functionalities while CSSS can only support Access.

Another field that is steadily growing is that of grammar compressed texts (e.g. see the survey [22] and a practical algorithm [21]). Some elegant results can decode a substring of length m in $O(m + \lg n)$ time [2], where n is the length of the uncompressed text, and there is a matching lower bound for the logarithmic cost [35]. In general, these methods can potentially compress better than the CSSS described in this survey, but finding the optimal grammar is NP-hard. The problem admits a logarithmic approximation factor [6,31], where the measure is the number of rules rather than the number of bits to represent the grammar.

Several other kinds of encodings have the property that each substring of the input text is translated into a substring of the corresponding encoded text (e.g. [3,23]), but they do not achieve high order entropy but still compress well in practice. One theoretical question is related to the encoding in [8] that shows how to store optimally a sequence of integers with constant-time random access to read and change one element. In our notation, the stated bound is $\lceil n\lg\sigma\rceil + O(1)$

bits, which can save $\Theta(n)$ bits over the raw representation in $n\lceil \lg \sigma \rceil$ bits. When comparing it to the bound of $nH_k + o(n \lg \sigma)$ for CSSS, the extra space is just $O(1)$ bits instead of $o(n \lg \sigma)$, but H_k can be much lower than $\lg \sigma$ for compressible text (and so $nH_k = o(n \lg \sigma)$). An interesting open problem is whether it is possible to combine the best of the two choices in some tradeoff that can blend the two opposite situations mentioned above.

Acknowledgments. I am in debt with Giuseppe Ottaviano for reading and commenting a preliminary version of this paper, and with Philip Bille, Gonzalo Navarro, Kunihiko Sadakane, Rossano Venturini for answering clarification requests, and Rossano for pointing out the open question mentioned at the end of the conclusions.

References

1. Barbay, J., He, M., Munro, J.I., Satti, S.R.: Succinct indexes for strings, binary relations and multilabeled trees. ACM Transactions on Algorithms 7(4), 52 (2011)
2. Bille, P., Landau, G.M., Raman, R., Sadakane, K., Satti, S.R., Weimann, O.: Random access to grammar-compressed strings. In: SODA, pp. 373–389 (2011)
3. Brisaboa, N.R., Ladra, S., Navarro, G.: Directly addressable variable-length codes. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 122–130. Springer, Heidelberg (2009)
4. Brisaboa, N.R., Cánovas, R., Claude, F., Martínez-Prieto, M.A., Navarro, G.: Compressed string dictionaries. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 136–147. Springer, Heidelberg (2011)
5. Burrows, M., Wheeler, D.: A block sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation (1994)
6. Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., Shelat, A.: The smallest grammar problem. IEEE Transactions on Information Theory 51(7), 2554–2576 (2005)
7. Demaine, E.D., López-Ortiz, A.: A linear lower bound on index size for text retrieval. J. Algorithms 48(1), 2–15 (2003)
8. Dodis, Y., Patrascu, M., Thorup, M.: Changing base without losing space. In: STOC, pp. 593–602 (2010)
9. Ferragina, P., Venturini, R.: A simple storage scheme for strings achieving entropy bounds. Theor. Comput. Sci. 372(1), 115–121 (2007)
10. Fraenkel, A.S., Kleinb, S.T.: Robust universal complete codes for transmission and compression. Discrete Applied Mathematics 64(1), 31–55 (1996)
11. Fredman, M.L., Saks, M.E.: The cell probe complexity of dynamic data structures. In: STOC, pp. 345–354 (1989)
12. Fredriksson, K., Nikitin, F.: Simple random access compression. Fundam. Inform. 92(1-2), 63–81 (2009)
13. Gál, A., Miltersen, P.B.: The cell probe complexity of succinct data structures. Theor. Comput. Sci. 379, 405–417 (2007)
14. Golynski, A.: Optimal lower bounds for rank and select indexes. Theor. Comput. Sci. 387, 348–359 (2007)
15. González, R., Navarro, G.: Statistical encoding of succinct data structures. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 294–305. Springer, Heidelberg (2006)

16. Grossi, R.: A quick tour on suffix arrays and compressed suffix arrays. Theor. Comput. Sci. 412(27), 2964–2973 (2011)
17. Grossi, R., Raman, R., Rao, S.S., Venturini, R.: Dynamic compressed strings with random access. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M., Peleg, D. (eds.) ICALP 2013, Part I. LNCS, vol. 7965, pp. 504–515. Springer, Heidelberg (2013)
18. Hon, W.-K., Shah, R., Vitter, J.S.: Compression, indexing, and retrieval for massive string data. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 260–274. Springer, Heidelberg (2010)
19. Jansson, J., Sadakane, K., Sung, W.K.: Cram: Compressed random access memory. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP 2012, Part I. LNCS, vol. 7391, pp. 510–521. Springer, Heidelberg (2012)
20. Kosaraju, R., Manzini, G.: Compression of low entropy strings with Lempel-Ziv algorithms. SIAM Journal of Computing 29(3), 893–911 (1999)
21. Larsson, N.J., Moffat, A.: Offline dictionary-based compression. In: Data Compression Conference, pp. 296–305 (1999)
22. Lohrey, M.: Algorithmics on slp-compressed strings: A survey. Groups Complexity Cryptology 4(2), 241–299 (2012)
23. Manber, U.: A text compression scheme that allows fast searching directly in the compressed file. ACM Trans. Inf. Syst. 15(2), 124–136 (1997)
24. Manzini, G.: An analysis of the Burrows-Wheeler transform. Journal of the ACM 48(3), 407–430 (2001)
25. Munro, J.I.: Tables. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180, pp. 37–42. Springer, Heidelberg (1996)
26. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comput. Surv. 39(1) (2007)
27. Ottaviano, G., Grossi, R.: Fast compressed tries through path decompositions. In: ALENEX, pp. 65–74 (2012)
28. Patrascu, M., Viola, E.: Cell-probe lower bounds for succinct partial sums. In: Charikar, M. (ed.) SODA, pp. 117–122. SIAM (2010)
29. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding *k*-ary trees, prefix sums and multisets. ACM Transactions on Algorithms 3(4) (2007)
30. Raman, R., Rao, S.S.: Succinct representations of ordinal trees. In: Brodnik, A., López-Ortiz, A., Raman, V., Viola, A. (eds.) Munro Festschrift 2013. LNCS, vol. 8066, pp. 319–332. Springer, Heidelberg (2013)
31. Rytter, W.: Application of lempel-ziv factorization to the approximation of grammar-based compression. Theor. Comput. Sci. 302(1-3), 211–222 (2003)
32. Sadakane, K.: Personal communication (2012)
33. Sadakane, K., Grossi, R.: Squeezing succinct data structures into entropy bounds. In: Proc. of the 17th ACM-SIAM SODA, pp. 1230–1239 (2006)
34. Sadakane, K., Grossi, R.: Squeezing succinct data structures into entropy bounds. In: SODA, pp. 1230–1239. ACM Press (2006)
35. Verbin, E., Yu, W.: Data structure lower bounds on random access to grammar-compressed strings. In: Fischer, J., Sanders, P. (eds.) CPM 2013. LNCS, vol. 7922, pp. 247–258. Springer, Heidelberg (2013)
36. Witten, I.H., Moffat, A., Bell, T.C.: Managing Gigabytes: Compressing and Indexing Documents and Images. Morgan Kaufmann Publishers (1999)
37. Witten, I.H., Neal, R.M., Cleary, J.G.: Arithmetic coding for data compression. Commun. ACM 30(6), 520–540 (1987)
38. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory 24(5), 530–536 (1978)