Squeezing Succinct Data Structures into Entropy Bounds

Kunihiko Sadakane* Roberto Grossi†

Abstract

Consider a sequence S of n symbols drawn from an alphabet $\mathcal{A} = \{1, 2, \ldots, \sigma\}$, stored as a binary string of $n \log \sigma$ bits. A *succinct* data structure on S supports a given set of primitive operations on S using just $f(n) = \mathrm{o}(n \log \sigma)$ extra bits. We present a technique for transforming succinct data structures (which do not change the binary content of S) into *compressed* data structures using $nH_k + f(n) + \mathrm{O}(n(\log \sigma + \log\log_\sigma n + k)/\log_\sigma n)$ bits of space, where $H_k \leq \log \sigma$ is the kth-order empirical entropy of S. When $k + \log \sigma = \mathrm{o}(\log n)$, we improve the space complexity of the succinct data structure from $n \log \sigma + \mathrm{o}(n \log \sigma)$ to $nH_k + \mathrm{o}(n \log \sigma)$ bits by keeping S in compressed format, so that any substring of $\mathrm{O}(\log_\sigma n)$ symbols in S (i.e. $\mathrm{O}(\log n)$ bits) can be decoded on the fly in constant time. Thus, the time complexity of the supported operations does not change asymptotically. Namely, if an operation takes $t(n)$ time in the succinct data structure, it requires $\mathrm{O}(t(n))$ time in the resulting compressed data structure. Using this simple approach we improve the space complexity of some of the best known results on succinct data structures We extend our results to handle another definition of entropy.

1 Introduction

Space-efficient data structures are useful in computationally demanding applications that require all data to reside in main memory, as is the case of search engines and portable computing. In this context, it is important to keep data in compressed format while allowing fast retrieval and update. Previous work in this direction led to a plethora of algorithmic solutions for data structures that are referred to as succinct, compact or compressed. These solutions share the idea of efficiently supporting a repertoire of basic functionalities on a succinct encoding of the data set, *without* the need to decode data entirely (only a negligible part is decoded). They combine the benefits of data compression with those of sophisticated search data structures. Examples of succinct encodings of this kind involve trees [3, 15, 27, 30, 29], graphs [21, 6], sets and dictionaries [5, 7, 16, 32, 35, 34], permutations and functions [28, 31], text indexes [11, 19, 36, 37], prefix or range sums [33], polynomials and others [13]. In general, each of these succinct encodings can be seen as a sequence S of n symbols drawn from an alphabet $\mathcal{A} = \{1, 2, \ldots, \sigma\}$, so encoding S uses $n \log \sigma$ bits.

We model a *succinct* data structure as operating in the uniform-cost word RAM with word size $b = \Theta(\log n)$; in particular, a single access to the memory can read or write a word of b bits. The data structure supports a given set of primitive operations on S using just $f(n) = \mathrm{o}(n \log \sigma)$ extra bits of space (i.e. $\lceil f(n)/b \rceil$ words of memory). Typically, the $f(n)$ bits are employed to store a set of directories that permit a fast access to data and require the decoding of few words in S. Hence, the space occupancy of succinct data structures can be expressed as $n \log \sigma + f(n) = n \log \sigma + \mathrm{o}(n \log \sigma)$ bits. The design of succinct data structures is mainly driven by the need to attain low space occupancy, and careful analysis is directed to evaluate the term $f(n)$ that models the number of extra bits. Many primitive operations are supported in this way, and we discuss a couple of significant examples in Sections 1.2 and 1.3, casting their operations into the following classification of the state of the art:

1. Those operations that probe the explicit sequence S and some auxiliary directories using $f(n)$ extra bits.
2. Those operations that replace the original encoding S' by an equivalent sequence S, and probe some auxiliary directories that use $f(n)$ extra bits.

Note that S itself is already a form of compression.

1.1 Matter of Investigation and Results

In this context, the purpose of our research investigation can be summarized in one question: *"What if we represent S implicitly by a shorter binary sequence, ξ, that contains the same information as S does?"*

If S is already a form of compression, it is unclear how much benefit we can expect from using ξ. Although the ultimate lower bound to space analysis is the Kolmogorov complexity [23], defined in terms of the

*Department of Computer Science and Communication Engineering, Kyushu University Hakozaki 6-10-1, Higashi-ku, Fukuoka 812-8581, Japan (sada@csce.kyushu-u.ac.jp). Work supported in part by the Grant-in-Aid of the Ministry of Education, Science, Sports and Culture of Japan.

†Dipartimento di Informatica, Università di Pisa, Largo Bruno Pontecorvo 3, 56127 Pisa, Italy (grossi@di.unipi.it). Work supported in part by MIUR of Italy.

SODA '06, January 22-26, Miami, FL
 ISBN 0-89871-605-5/06/01

size of the smallest program that can generate S, its undecidability naturally motivates the definition of "sub-optimal" measures in the space complexity. For instance, entropy is widely adopted for texts [8]; finite set model and encoding of integers are suitable for dictionaries [7]; counting arguments (such as the Catalan number and Tutte's bounds) are employed for parenthesis representations of trees and graphs [26], and so on. These are widely accepted lower bounds for evaluating the nearly optimal space occupancy of succinct data structures.

Depending on the succinct data structure at hand, we have room for improvement in ξ by choosing a suitable measure that is better than what offered by the state of the art. We remark that the importance of this kind of investigation is not merely theoretical, since theory is the basis for studying the possibilities and the limitations of practical methods for succinct data structures, analogously to what happened to information theory in data compression [8].

In this paper, we give an answer to the above question by showing how to improve the best known space bounds in the literature while preserving the same asymptotical time bounds. We require that the operations supported by the succinct data structures do not change the binary content of S.

We propose a technique that transforms a given succinct data structure on S that uses $f(n)$ extra bits, into a *compressed* data structure with entropy bounds,

$$nH_k + f(n) + \mathrm{O}\left(\frac{n(\log\sigma + \log\log_\sigma n + k)}{\log_\sigma n}\right) \quad (1.1)$$

bits, where $H_k \leq \log\sigma$ is the kth-order empirical entropy of the sequence S (see [22]). The idea is widely usable since it is conceptually simple, and it is the first to attain high-order entropy for the class of succinct data structures discussed below, using Ziv-Lempel compression [38] (LZ78) and other properties. Instead of keeping S explicitly stored, we replace S by a compressed representation ξ, so that any substring $S[i \dots i+r]$ can be recovered from ξ in $\mathrm{O}(1 + r/\log_\sigma n)$ time, and so any word of S can be recovered from ξ in $\mathrm{O}(1)$ time. Note that ξ is actually the encoding of a special succinct data structure that uses no more bits than S (plus lower order terms) and supports a primitive operation that decodes an arbitrary substring of S.

When $k + \log\sigma = \mathrm{o}(\log n)$ in equation (1.1), we improve the space complexity of the succinct data structure by replacing S with ξ, reducing the number of bits from $n\log\sigma + \mathrm{o}(n\log\sigma)$ to $nH_k + \mathrm{o}(n\log\sigma)$. Note that nH_k can be much smaller than $n\log\sigma$ as discussed in the examples of Sections 1.2 and 1.3. When a supported operation requires a certain memory word $\omega \in S$, we introduce an intermediate layer that applies the substring decoding operation for recovering the word ω from ξ in constant time. Thus, the time complexity of the supported operations do not change asymptotically; namely, if an operation takes $t(n)$ time in the succinct data structure, it requires $\mathrm{O}(t(n))$ time in the resulting compressed data structure.

Note that the bound on H_k holds for S in equation (1.1). According to our classification (cf. point 2), when S replaces the original encoding S', we may attempt to compress S' directly by means of ξ (if a word in S can be obtained from S' in constant time). In this case, the bound on H_k in (1.1) refers to S'. If this is not the case, $H_k = H_k(S)$ has to be related to the kth-order empirical entropy of the original encoding S'.

Using our simple approach we improve the space complexity of some of the best known results on succinct data structures (setting $\sigma = 2$ and $k = \mathrm{O}(\log\log n)$), while preserving the constant-time complexity of their supported operations. We discuss these implications in Sections 1.2 and 1.3, along with an alternative application of our ideas on replacing S by ξ, for a measure of entropy based on gap encoding. We refer the reader to Section 3 for a comparison of our results with related work on entropy-compressed data structures.

1.2 Succinct Dictionaries

Let us examine first the application of our general method to the problem of *succinct dictionaries*, improving the best known bounds. Consider an ordered set $\hat{S} \subset U = \{1, 2, \dots, n\}$ of $m \leq n/2$ elements (otherwise, we take the complement of $\hat{S}$). According to our classification, the first approach is that of representing $\hat{S}$ by a bit-vector $S[1 \dots n]$, such that $S[i] = 1$ if the ith element of U belongs to $\hat{S}$, and $S[i] = 0$ otherwise (here, $\sigma = 2$). In addition to S, the set of supported operations use $f(n) = \mathrm{o}(n)$ bits for directories implementing the following constant-time functions on S, where $x = 0, 1$ (see Jacobson [21] and Clark and Munro [25]):

- $rank_x(S, i)$ returns the number of occurrences of x in $S[1 \dots i]$, where $1 \leq i \leq n$;
- $select_x(S, i)$ returns the position of the ith occurrence of x in $S[1 \dots n]$, where $1 \leq i \leq m$.

Note that checking the membership of element i in $\hat{S}$ can be easily performed by the above two primitives. This is equivalent to verifying whether $S[i] = 1$ (or $S[i] = 0$ if we complemented $\hat{S}$). The *rank* and *select* functions can be extended as follows [27]:

- $rank_p(S, i)$ returns the number of occurrences of pattern p up to the position i, where p is for example "01" and $|p| = \mathrm{O}(1)$.
- $select_p(S, i)$ returns the position of ith occurrence of pattern p (where $|p| = \mathrm{O}(1)$).

The second approach in our classification is based on the observation that the above representation of S has much redundancy when m is small. Since there are only $\binom{n}{m}$ different sets with m 1's, we have that $\mathcal{B}(m,n) = \lceil \log \binom{n}{m} \rceil \leq n$ is an information-theoretic lower bound on the number of bits required to store $\hat{S}$. Brodnik and Munro [7] were the first to show how to get this bound. Pagh [32] discussed how to implement the *rank* function. Raman, Raman and Rao [34] proposed data structures for *rank* and *select* queries for S called *fully indexable dictionary* (FID). In this case, the representation S becomes a binary string of $\mathcal{B}(m,n)$ bits, and $f(n) = \mathrm{O}(n \log\log n / \log n)$, yielding a nearly optimal size. This represents the best bound known for the problem.

We can apply our technique to both approaches and get the *same* improved bounds. Setting $\sigma = 2$ and $k = \mathrm{O}(\log\log n)$ in equation (1.1), we obtain a space bound of $nH_k + \mathrm{O}(n \log\log n / \log n)$ bits, still supporting *rank* and *select* in constant time. To appreciate the improvement, note that $\mathcal{B}(m,n) \approx m \log \frac{n}{m} + (n-m) \log \frac{n}{n-m} = nH_0$. Since $H_k \leq H_{k-1} \cdots \leq H_0$, our implementation (which uses FIDs) improves the space bounds of FIDs themselves! We call our dictionaries *entropy-bound indexable dictionaries* (EIDs). Note that the auxiliary data structures in FIDs can also be employed if the corresponding sequence S is stored in an uncompressed form, i.e., represented by a bit-vector of length n. The FIDs divide the bit-vector into segments of $\frac{1}{2}\log n$ bits and store them in a compressed form, along with the pointers to the segments.

This improvement can be drastic as it depends on the distribution of the m 1s in the uncompressed S. For example, take $m = n/2$. We have $\mathcal{B}(m,n) \approx n$ bits independently of the distribution of the 1s. However, if they occur contiguously in $S[i \ldots i+m-1]$, or in alternating position $S[i]$, $S[i+2]$, $\ldots$, $S[i+2(m-1)]$, we expect to represent them in $nH_1 = \mathrm{O}(\log n)$ bits, with an exponential improvement. Unfortunately, there is a limit on the maximally attainable improvement because of the extra term $f(n) = \mathrm{O}(n \log\log n / \log n)$. Indeed, Miltersen [24] proved that $f(n) = \Omega(n \log\log n / \log n)$ for constant-time *rank* and *select* when S is read-only. Nevertheless, EIDs still obtain a better bound of $\mathrm{O}(n \log\log n / \log n)$ in these cases by equation (1.1), while FIDs take $n + \mathrm{o}(n)$ bits. [1]

When $m \ll n$, we cannot expect big improvements with a small value of k. It is more suitable to define entropy in terms of the gap bounds. For an integer $g > 0$, let $\delta(g) = 1 + \lfloor \log g \rfloor$ be the minimum number of bits to represent g in binary. Also, let s_i denote the ith smallest element of $\hat{S}$, where $s_0 = 0$. We define

$$gap(\hat{S}) = \sum_{i=1}^{m} \delta(s_i - s_{i-1}), \tag{1.2}$$

and we consider only the case for which $gap(\hat{S}) < n$ (which is true when $m \ll n$) because otherwise we can use EIDs. Then, we can obtain a variant of EIDs based on gap encoding that uses $gap(\hat{S}) + \mathrm{O}(n \log\log n / \log n)$ bits. Since it can be proved [18] that $gap(\hat{S}) \leq \mathcal{B}(m,n)$, we obtain another improvement. In summary, we obtain the first space bound of

$$\min\{nH_k, gap(\hat{S})\} + \mathrm{O}(n \log\log n / \log n) \tag{1.3}$$

bits for succinct dictionaries supporting constant-time *rank* and *select*. The first argument of $\min\{\cdot,\cdot\}$ in equation (1.3) is more pertinent to dense sets while the second is more suitable to sparse sets.

We believe that our technique can improve other bounds on succinct dictionaries in the literature. An example of improvement is performing $rank_x$ and $select_x$ for any $x \in \mathcal{A}$ as shown by the *wavelet trees* of Gupta, Grossi and Vitter [17] and their alternative implementation by Ferragina, Mäkinen, Manzini and Navarro [10]. We improve again their space from $nH_0 + \mathrm{O}(n \log\log n / \log_\sigma n)$ to $nH_k + \mathrm{O}(n \log\log n / \log_\sigma n)$ bits while preserving the same time bounds, where $\sigma = |\mathcal{A}| = \mathrm{polylog}(n)$ and $k = \mathrm{O}(\log\log n)$. In general, we are able to preserve the time complexity of the primitives supported by a succinct dictionary, provided that they do not change the underlying sequence S.

1.3 Succinct Representation of Trees

Another example of application of our technique is the *succinct representation of trees*. Consider a rooted ordered tree with n nodes. It is well known that the tree is represented by a balanced parenthesis sequence S of $2n$ bits as follows. During a preorder traversal of the tree, write an open parenthesis when a node is visited, then traverse all subtrees of the node, and write a close parenthesis. Any node in the tree is represented by a pair of open and close parentheses "$(\cdots)$". We use the position of the open parenthesis in S to indicate the node. This is an interesting case as the two approaches in our classification coincide, both yielding nearly $2n$ bits. Indeed we cannot compress this sequence by FIDs because there are n open parentheses and n close parentheses, implying $\mathcal{B}(n,2n) \approx 2n$. The best bounds use $2n + \mathrm{o}(n)$ bits for supporting several primitive operations that navigate in the encoded tree (e.g. [3, 15, 21, 27, 29, 27, 30, 34, 35]).

[1] Note that our technique cannot be applied to indexable dictionaries in [34], which support a restricted form of $rank_1$ and $select_1$. This makes a difference as constant-time predecessor queries cannot be performed in these dictionaries.

In this context, our approach compresses S in $\min\{2nH_k, gap(\hat{S})\} + \mathrm{O}(n \log\log n / \log n)$ bits as in (1.3). However, H_k refers to the kth-order empirical entropy of the parentheses in S and $gap(\hat{S})$ should be interpreted by taking the)s in S as elements of a set $\hat{S} \subseteq [1 \dots 2n]$.

Our entropy bound takes into consideration the skewness of the shape of the encoded tree. For example, the degenerate case of a tree which is unbalanced to its left, gives the highly-repetitive sequence of parentheses "((((⋯()())⋯())", whose symbols can be predicted by looking at the $k = 2$ preceding symbols; hence, we expect to compress it in $2nH_2 = \mathrm{O}(\log n)$ bits. (Here the higher-order entropy H_k refers to the sequence of $2n$ parentheses.) This is especially effective to compress balanced parenthesis sequences representing a suffix tree because the latter contains many similar subtrees which are compressible.

It is worth noting that we can still support the following tree navigational operations in constant time [14, 15, 26, 27], where $f(n) = \mathrm{o}(n)$ in equation (1.1): finding the first child, the next sibling, the parent, and the leftmost and the rightmost descendants, computing the number of descendants, the number of leaves of a subtree, the depth of a node, and the preorder and the postorder of a node, etc. All the queries above take constant time (where $f(n) = \mathrm{O}(n \log\log n / \log n)$, see [14, 27]). In addition to these operations, we can also support the following constant-time operation using $f(n) = \mathrm{O}(n \log\log n / \sqrt{\log n})$ bits:

- $anc(S, v, d)$ returns the level-ancestor of node v that is of depth d [31].

Once again, we believe that our approach has a wide applicability, for example, in encoding important subclasses of trees in less than $2n + \mathrm{o}(n)$ bits.

2 Entropy-Bound Indexable Dictionary (EID)

We now describe our scheme for transforming succinct data structures that access (but do not modify) a sequence S of n symbols drawn from an alphabet $\mathcal{A} = \{1, 2, \dots, \sigma\}$, into compressed data structures with entropy bounds. We recall that S can be seen as a binary string of length $n \log \sigma$. Our purpose is to decode $w = \frac{1}{2} \log_\sigma n$ consecutive symbols (i.e. $w' = \frac{1}{2} \log n$ bits) of S from its compressed format ξ in constant time, for any arbitrary position. In this way, we can support all the primitive operations defined on S by the succinct data structure at hand. We reuse all the machinery from the succinct data structure; when a portion of S is needed, we plug in our method to provide the required data from ξ. This simple idea is very effective for improving the previously known bounds.

We illustrate our technique with a concrete example, thus proposing *entropy-bound indexable dictionaries* (EIDs) whose properties are given in Theorem 2.1.

THEOREM 2.1. *There exists a succinct data structure for storing a string $S[1 \dots n]$ on alphabet $\mathcal{A} = \{1, 2, \dots, \sigma\}$ in*

$$nH_k + \mathrm{O}\left(\frac{n(\log\sigma + \log\log_\sigma n + k)}{\log_\sigma n}\right) \qquad (2.4)$$

bits for any $k \geq 0$, where H_k is the kth-order empirical entropy of S, such that any substring $S[i \dots i + \mathrm{O}(\log_\sigma n)]$ of $\mathrm{O}(\log n)$ bits can be decoded in $\mathrm{O}(1)$ time on the word RAM, for $1 \leq i \leq n$.

COROLLARY 2.1. *The above data structure has size $nH_k + \mathrm{O}(n \log\log n / \log n)$ bits when $\sigma = \mathrm{O}(1)$ and $k = \mathrm{O}(\log\log n)$; in general, it has size $nH_k + \mathrm{o}(n \log\sigma)$ when $k + \log\sigma = \mathrm{o}(\log n)$.*

Due to lack of space, we refer the reader to Sections 1.2 and 1.3 for the improvements on succinct dictionaries and succinct representation of trees that follow from Theorem 2.1. In Section 2.1, we briefly review the parsing of Ziv-Lempel compression [38], which is at the heart of our method, and the LZ-trie resulting from that parsing. To support our substring decoding query, we need to decode some paths in the LZ-trie. We introduce new succinct data structures for supporting this task in Section 2.2, and analyze their space complexity. In Section 2.3, we describe the substring decoding query algorithms and discuss their time complexity. In Section 2.4, we describe an alternative way to compress S for the dictionary problem, using the *gap encoding* according to equation (1.2), so as to get the bound in (1.3).

2.1 Ziv-Lempel Compression

Ziv-Lempel compression [38], or LZ78, is a data compression scheme for strings. When applied to $S = S[1 \dots n]$, the algorithm works as follows. First initialize a trie T as empty, the current position $p = 1$ in S, and the number of phrases $c = 0$. Then, parse S into phrases from left to right as follows. Find the longest string, $t \in T$, that appears as a prefix of $S[p \dots n]$. Obtain the phrase $s = S[p \dots p + |t|] = t \cdot S[p + |t|]$ to be inserted into T, and determine t's index in T (which is the identifier of the node in T storing t). Set $c = c+1$, and output t's index (encoded in $\lceil \log c \rceil$ bits) followed by symbol $S[p + |t|]$ (encoded in $\lceil \log\sigma \rceil$ bits). Set $p = p + |t| + 1$ and repeat the parsing for the next phrase. The resulting trie T is called an LZ-*trie*, denoting by c the number of phrases generated by algorithm LZ78 (and thus the number of nodes in T).

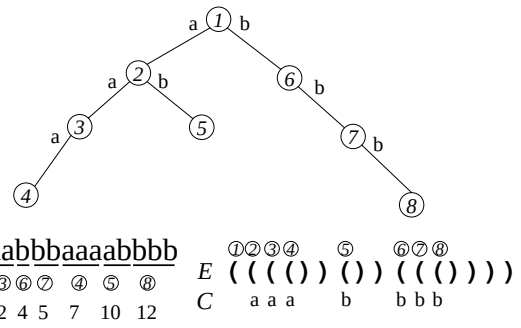

Figure 1: Encoding of LZ-trie for $S =$ aaabbbaaaabbbb with the first suite of arrays R, P, E, and C.

LEMMA 2.1. (ZIV AND LEMPEL [38]) *The number of phrases in* LZ78 *satisfies*

$$\sqrt{n} \le c \le n/\log_\sigma n. \tag{2.5}$$

For strings generated from a stationary ergodic source, we have $\frac{c\log c}{n} \to H$ where H is the entropy of the source [8]. On the other hand, the output size of LZ78 is at most $c(\log c + \log\sigma)$. Therefore the compression ratio of LZ78 achieves the optimal asymptotically. We use Lemma 2.3 from [22] for bounding c in terms of the kth-order empirical entropy, H_k, of the string S:

LEMMA 2.2. (KOSARAJU AND MANZINI [22]) *Let s_1, s_2, ..., s_c denote a parsing of the string S, such that each phrase appears at most M times. For any $k > 0$, the number of phrases satisfies*

$$c\log c \le |S|H_k + c\log\frac{|S|}{c} + c\log M + \Theta(kc + c) \tag{2.6}$$

2.2 Succinct Data Structures for Substring Decoding

Our data structures are built around the LZ-trie T, which is obtained from running the LZ78 parsing for S as described in Section 2.1, where c is the number of phrases. Let $p_1, p_2, \ldots, p_c$ denote the starting positions of the phrases in S (where $1 = p_1 < p_2 < \cdots < p_c \le n$). We renumber the indices assigned to the phrases by LZ78, so that they coincide with the preorder numbering of the corresponding nodes in the LZ-trie T. (The node corresponding to a phrase is defined as the node storing that phrase in T.) We denote by r_j the "preorder" index of phrase j, for $1 \le j \le c$.

We succinctly represent the LZ-trie T by storing two suites of arrays. The first suite contains the following arrays, illustrated by the example shown in Figure 1 and summarized in Table 1.

- $R[1\ldots c]$ contains the preorder indices $r_1, r_2, \ldots, r_c$ of the phrases in S generated by LZ78. Note that they are a permutation of $1, 2, \ldots, c$. (We actually store a subset of these indices.)
- $P[1\ldots c]$ contains the starting positions p_1, p_2, ..., p_c of the phrases in S. It is stored as a subset of c elements from the universe $1\ldots n$, using a FID of $\mathcal{B}(c,n) + \mathrm{O}(n\log\log n/\log n) \le c\log\frac{n}{c} + 1.44c + \mathrm{O}(n\log\log n/\log n)$ bits of space, since $(n-c)\log\frac{n}{n-c} \le 1.44c$. (We refer the reader to Section 1.2 for the operations supported by FIDs.)
- $E[1\ldots 2c]$ contains the balanced-parenthesis sequence representing the tree shape of the LZ-trie T. It is stored as a binary sequence in $2c$ bits augmented with auxiliary directories to navigate in the tree, using further $\mathrm{O}(c\log\log c/\sqrt{\log c}) = \mathrm{o}(c)$ bits. (We refer the reader to Section 1.3 for the primitive operations supported by the tree encoding E.)
- $C[2\ldots c]$ contains the symbols of $\mathcal{A}$ labeling the edges of the LZ-trie T, when traversed in preorder. In particular, $C[i]$ is the label of the edge (u,v), where u is v's parent and v is the node in T represented by the ith open parenthesis "(" in E. ($C[1]$ is not defined because the first node has no parent.) Array C stores c symbols in $c\log\sigma$ bits.

The purpose of the first suite of arrays is to support navigational operations in the LZ-trie T, retrieve its edge labels, and map positions and phrases of S into T's nodes. However, we need the second suite of arrays to decode arbitrary substrings of $w = \frac{1}{2}\log_\sigma n$ symbols using T, namely, $w' = \frac{1}{2}\log n$ bits. Here, we set up the following machinery, as summarized in Table 2.

We define *short phrases* as those which consist of less than w symbols, and *long phrases* as the opposite. For short phrases, we introduce a succinct data structure for decoding consecutive short phrases. For long phrases, we need succinct data structures for decoding the labels in the length-w path from v upward to the root, for any given node $v \in T$. (If v is at depth $\bar{w} < w$, then $\bar{w}$ symbols are returned.) For this, we define three types of nodes: jump nodes, macro nodes, and micro nodes. (We adopt these terms from Bender and Farach-Colton [2].) We define a *jump* node as a node v such that v has at least $w/2 = \frac{1}{4}\log_\sigma n$ descendants (including v itself) while every child of v has less than $w/2$ descendants. There exist at most $4c/\log_\sigma n$ jump nodes. A *macro* node is a node which is an ancestor of a jump node (including the jump node). Finally, *micro* nodes are the non-macro nodes. We also define a *micro tree* as a maximal connected component of micro nodes. Each micro tree contains less than $w/2$ nodes (otherwise it would contain a macro node). Also, if a node belongs to a micro tree Q, all of its descendents belong to Q.

array	$R[1\dots c]$ (*)	$P[1\dots c]^{\S 1.2}$	$E[1\dots 2c]^{\S 1.3}$	$C[2\dots c]$
#bits	see Lemma 2.3	$\mathcal{B}(c,n)+\mathrm{O}(n\log\log n/\log n)$	$2c+\mathrm{o}(c)$	$c\log\sigma$

(*) *Array R is actually shorter than declared. See Lemma 2.3 for the actual space occupancy.*

Table 1: First suite of arrays for representing the LZ-trie T and their space occupancy in bits. The array tagged with §1.2 is a FID and the one tagged with §1.3 is a succinct representation of trees.

We define a tree $\tilde{T}$ as the induced subtree consisting of all macro nodes of the LZ-trie T. All leaves of $\tilde{T}$ are jump nodes. For representing $\tilde{T}$, we store the following.

- $\tilde{E}[1\dots 2c]$ contains the balanced-parenthesis sequence representing $\tilde{T}$. It is stored as a binary sequence in at most $2c$ bits augmented with auxiliary directories of $\mathrm{O}(c\log\log c/\log c)$ bits.
- $\tilde{C}[1\dots c]$ contains edge labels of $\tilde{T}$, when traversed in preorder.
- $F[1\dots c]$ stores flags to show which nodes in T are macro nodes. A node in T whose preorder number is i is a macro node if and only if $F[i]=1$. The preorder number of the corresponding node in $\tilde{T}$ is computed by $\mathit{rank}(F,i)$.

For decoding a path in $\tilde{T}$, we use the following.

- $B[1\dots c]$, $W_B[1\dots 4c/\log_\sigma n-1]$: We consider the length-w path for each *branching* node of $\tilde{T}$. There are at most $4c/\log_\sigma n-1$ such nodes. We mark all branching nodes by using a bit-vector $B[1\dots c]$, which is stored as a FID in $c+\mathrm{O}(c\log\log c/\log c)$ bits. The node in $\tilde{T}$ with preorder number v is marked as branching node if and only if $B[v]=1$. We store the labels along the length-w paths leading to branching nodes (from the root) into an array W_B, where the labels on the path for v are stored in entry $W_B[\mathit{rank}(B,v)]$, using less than $2c\log\sigma$ bits. (For *unary* nodes, with exactly one child, we will decode paths by using the array $\tilde{C}$.)

As previously mentioned, we also need a succinct data structure for decoding consecutive short phrases. If there is a region in S of $r>1$ consecutive phrases, all of them shorter than w, we replace the storage of the indices in R with the direct storage of the orderly concatenation of these phrases. We define a *dense region* to be a region with the largest number $r>1$ of consecutive phrases, provided that their total length does not exceed w. We also impose the constraint that any two dense regions must be separated by a (either short or long) phrase. We guarantee this property, uniquely identifying the dense regions in S, by scanning S from left to right in greedy fashion.

Let d denote the number of dense regions thus identified. We employ the following data structures.

- $D[1\dots c]$, $W_D[1\dots d]$: We use a bit-vector $D[1\dots c]$ to mark the dense regions in S, with $r>1$ consecutive short phrases. Namely, the jth phrase is in a dense region if and only if $D[j]=1$. Note that by our constraint, two regions cannot be consecutive, so there is at least a 0 in D separating them. Again, D is stored as a FID in $c+\mathrm{O}(c\log\log c/\log c)$ bits. In array $W_D[1\dots d]$, we store the length-w substrings of S that contain the dense regions in S (since a dense region can be shorter than w).

Note that we do *not* store anymore in R the preorder indices of the short phrases contained in the dense regions of S. For the remaining phrases (i.e. the short phrases not contained in any dense region and the long phrases), we store their preorder indices in R as usual. Let R' denote the resulting array, which contains a subset of the indices in R. We can simulate the access to R as $R[j]=R'[\mathit{rank}_0(D,j)]$ where j is the preorder index of a phrase not contained in a dense region. In the following, we will use the notation R in place of R', so that its length can be shorter than that given in Table 1.

LEMMA 2.3. *Arrays R and $W_D[1\dots d]$ use together at most $c\log c$ bits.*

Proof. Recall that W_D stores the dense regions of S, allocating $w'=\frac{1}{2}\log n$ bits per region. We show that the space bound of W_D is smaller than the number of bits originally allocated in R for storing the preorder indices of the short phrases that are currently in the dense regions. (Note that we do not store anymore their preorder indices.) Recall that a dense region consists of r phrases, for some $r>1$. We would use $r\log c$ bits if we had to store these phrases in R. By Lemma 2.1, we know that $c\geq\sqrt{n}$ in equation (2.5). Therefore, $r\log c>\frac{1}{2}\log n$ because $r>1$. This implies that $w'<r\log c$, meaning that the short phrases in the dense region uses less bits than storing their $r>1$ preorder indices in R. Hence the claim follows.

Finally, we need a lookup table for the micro trees, which are the subtrees of T rooted at the children of the macro nodes. Each micro tree, Q, contains at most

array	$B[1\dots c]^{\S 1.2}$	$W_B[1\dots 4c/\log_\sigma n - 1]$	$\tilde{E}[1\dots 2c]^{\S 1.3}$	$\tilde{C}[1\dots c]$
#bits	$c + \mathrm{o}(c)$	$2c\log\sigma$	$2c + \mathrm{o}(c)$	$c\log\sigma$
array	$D[1\dots c]^{\S 1.2}$	$W_D[1\dots d]$	$F[1\dots c]^{\S 1.2}$	$L[\alpha,\beta,\gamma]$
#bits	$c + \mathrm{o}(c)$	see Lemma 2.3	$c + \mathrm{o}(c)$	$\mathrm{O}(n^{3/4}\log^2 n)$

Table 2: Second suite of arrays for representing the LZ-trie T and their space occupancy in bits. The arrays tagged with §1.2 are FIDs while the array tagged with §1.3 is a succinct representation of trees.

$w/2 = \frac{1}{4}\log_\sigma n$ nodes by definition. Hence, Q can be represented by a segment, α, of at most $w/2$ symbols from array C and by the corresponding binary segment, β, of array E to encode its shape, with a total of $w/2 \times (\log\sigma + 2) \leq \frac{3}{4}\log n$ bits for any $\sigma \geq 2$. Hence, there are $\mathrm{O}(n^{3/4})$ distinct micro trees Q in T. Moreover, the preorder numbering of T implies that the nodes in Q are numbered consecutively. So, we can easily transform the global preorder number of a node $u \in Q$ into a local preorder number, γ, by subtracting the preorder number of the root of Q from u's preorder number. Using these facts, we use a lookup table, L, of $\mathrm{O}(n^{3/4}\log^2 n) = \mathrm{o}(n\log\log n/\log n)$ bits of space, such that entry $L[\alpha,\beta,\gamma]$ stores the labels on the path from $u \in Q$ to the root of Q (using at most w symbols). Each entry can be retrieved in constant time by a simple table lookup into L, using the proper segments α of C and β of E, along with the local preorder number γ of u.

We can sum up the sizes for all the data structures in the suites, as reported in Tables 1 and 2. The first suite, except R, occupies $\mathcal{B}(c,n) + c\log\sigma + 2c + \mathrm{o}(c) + \mathrm{O}(n\log\log n/\log n)$ bits of space. The second suite, except W_D, occupies $3c\log\sigma + 5c + \mathrm{o}(c) + \mathrm{O}(n^{3/4}\log^2 n)$ bits of space. By Lemma 2.3, arrays R and W_D together contribute for further $c\log c$ bits at most. Since $\mathcal{B}(c,n) \leq c\log\frac{n}{c} + 1.44c + \mathrm{O}(c)$, the total is $c\log c + c\log\frac{n}{c} + \mathrm{O}(c\log\sigma) + \mathrm{O}(n\log\log n/\log n)$ bits. From Lemma 2.1, we know that $c \leq n/\log_\sigma n$, and so we can bound $c\log\frac{n}{c} = \mathrm{O}(n\log\log_\sigma n/\log_\sigma n)$ and $c\log\sigma = \mathrm{O}(n\log\sigma/\log_\sigma n)$. Thus, the total space is upper bounded by $c\log c + \mathrm{O}(n(\log\sigma + \log\log_\sigma n)/\log_\sigma n)$ bits. We now apply Lemma 2.2 to the latter bound. In particular, we use equation (2.6) on the term $c\log c$, where $M = 1$ and $|S| = n$.

LEMMA 2.4. *The two suites of data structures reported in Tables 1 and 2 use a total of*

$$nH_k + \mathrm{O}\left(\frac{n(\log\sigma + \log\log_\sigma n + k)}{\log_\sigma n}\right)$$

bits of space for any $k \geq 0$, where H_k is the kth-order empirical entropy of the sequence S of n symbols drawn from an alphabet $\mathcal{A} = \{1, 2, \dots, \sigma\}$.

2.3 Substring Decoding Query Algorithms

We aim at decoding the substring $q = S[i\dots i+w-1]$ in constant time, where $w = \frac{1}{2}\log_\sigma n$ is the number of symbols in q, by providing in input its starting position i in S. It is simple to extend the query so as to decode a substring of $\mathrm{O}(w)$ consecutive symbols (i.e. $\mathrm{O}(\log n)$ bits) in constant time. Starting from the LZ78 parsing of S into phrases at positions $1 = p_1 < p_2 < \cdots < p_c \leq n$, we have set up a structure on these phrases in Section 2.2, so that we can see S as partitioned into dense regions, short phrases not contained in dense regions, and long phrases. Before showing how to compute q, we relate the structure of S's phrases to q.

LEMMA 2.5. *In the LZ78 parsing of S, no two dense regions are consecutive, and if two short phrases not contained in dense regions are consecutive their total length is at least w. Hence, substring q overlaps a constant number of dense regions, short phrases not contained in dense regions, and long phrases.*

We exploit the properties described in Lemma 2.5 for decoding substring q as follows:

1. Determine the phrase j containing $S[i]$ by computing $j = \mathit{rank}(P,i)$ and the position of the phrase $p_j = \mathit{select}(P,j)$ (note that $p_j \leq i$).
2. Determine the phrase j' containing $S[i+w-1]$ and its position $p_{j'}$, analogously to what done in Step 1.
3. Decode $q = S[i\dots i+w-1]$ using the phrases $j, j+1, \dots, j'$ and the data structures of Section 2.2, according to the following cases (distinguishable in constant time by comparing j to j' and p_j to $p_{j'}$):
   (a) q is entirely contained in phrase j (i.e. $j' = j$);
   (b) q is contained in two consecutive phrases j and $j' = j+1$, each phrase of at least w symbols;
   (c) q is contained in more than one phrase, one of which has length less than w.

Below we describe the algorithms for handling cases (a)–(c) in some detail. Note that there can

be a non-constant number of phrases in case (c), but Lemma 2.5 reduces them to a constant number of segments to decode. In the following, we identify a node v (in T or $\tilde{T}$) with its preorder number v in the tree.

Case (a): We first find the node $v \in T$ such that $S[p_j \dots i+w-1]$ is stored along the path from the root of T to v. (Hence, q is a suffix of the latter substring.) This is done by computing the level-ancestor query $v = anc(E, r_j, i+w-p_j)$ in constant time, where $r_j = R[j] = R'[rank_0(D, j)]$ is the node in T storing phrase j (and so r_j descends from v since $S[p_j \dots i+w-1]$ is a prefix of phrase j).

Consequently, we focus on the problem of decoding the labels on the length-w downward path leading to node v, since q is made up of these symbols. We first check if v is a micro node by examining $F[v]$. If so, v belongs to a micro tree, Q, rooted at a child of t, the lowest macro-node ancestor of v. We can identify t by finding the rightmost $F[t]$ such that $F[t] = 1$ and $t < v$ (found using *rank* and *select* on F). We can obtain a suffix of q as follows. Let α be the segment of the array C corresponding to Q, and β the segment of E corresponding to Q. Let $\gamma = v - t$ be the local preorder number of v in Q. A lookup at entry $L[\alpha, \beta, \gamma]$ returns the labels from the child of t to v. We can easily get the label from t to its child using C. If the number of decoded character is less than w, we obtain the character $C[t]$ on the edge connecting t to its parent. We continue decoding from t, which is a macro node, following the scheme described for v below.

If v is a macro node, we find the lowest ancestor s of v that is a branching node in $\tilde{T}$. This task is made easy by the observation that all the nodes are unary (in $\tilde{T}$) along the path connecting v to s. Hence, s corresponds to the position of the rightmost 1 in B such that $s < \tilde{v}$ because of the preorder numbering, where $\tilde{v}$ is the preorder of v in $\tilde{T}$ as computed by $rank(F, v)$. Hence, we can find s in constant time, and we can decode the path from the array $\tilde{C}$ as the symbols in $\tilde{C}[s+1 \dots \tilde{v}]$. If that path is shorter than w, we have to decode the upward path from the branching node. We obtain the labels on the path by accessing B and W_B, as discussed in Section 2.2. Finally, we obtain q as a length-w substring of concatenation of the paths decoded above.

Case (b): This case is similar to Case (a). Since q is included in just two long phrases and the w symbols of each phrase can be decoded in constant time, the claim holds.

Case (c): In this case, the number of phrases covering q may not be a constant. We use Lemma 2.5 to circumvent this drawback by decoding a constant number, ℓ, of segments. We parse q as $q = q_1 q_2 \cdots q_\ell$ on the fly, from left to right, such that each q_i is contained in a dense region, a short phrase not in a dense region, or a long phrase. (Actually, q_1 can be a suffix, q_ℓ can be a prefix, while the other q_is are equal to their companion phrases or regions.) We discussed how to deal with long phrases in case (a). Indeed, dealing with short phrases is also very similar to case (a), since we know that q_i is contained (or equal) to the phrase. We therefore focus on the case that q_i is in a dense region.

Without loss of generality, we consider q_1, namely, the beginning of q's parsing on the fly. Recall that phrase j contains the first symbol of q (hence, of q_1). We check if $D[j] = 1$, that is, phrase j is in a dense region. If not so, we know how to treat long and short phrases, as previously mentioned. Hence, let us assume that $D[j] = 1$. We obtain the index m of the dense region by executing $m = rank_{01}(D, j)$ since the beginning of a dense region is indicated by the pattern 01 in D. (The border case when the region is the leftmost in S can be easily handled.) All that remains is to decode the string from entry $W_D[m]$. By Lemma 2.5, we have no more than $\ell = O(1)$ decoding steps, yielding a constant-time cost for decoding q also in this case.

LEMMA 2.6. *For any sequence S of n symbols drawn from alphabet $\mathcal{A}$, we can decode any substring of S of $O(\log_\sigma n)$ consecutive symbols (i.e. $O(\log n)$ bits) in constant time on a word RAM, using the two suites of data structures reported in Tables 1 and 2.*

2.4 Gap Encoding Measure of Entropy

We show how to apply our general method as described in Section 1.1, to the special case of succinct dictionaries when $\sigma = 2$. We have the problem of encoding a set $\hat{S} \subset U = \{1, 2, \dots, n\}$ with m elements. We follow our general approach described in the Introduction. Let s_i be the ith smallest elements of $\hat{S}$, and $s_0 = 0$. We conceptually construct an implicit binary string $S[1 \dots n]$ such that $S[i] = 1$ for each $s_i \in \hat{S}$, and construct auxiliary data structures for *rank* and *select* in $O(n \log\log n / \log n)$ bits using a FID. Then we compress S by the concatenation of a binary encoding of the *gaps*, namely, the differences $(s_i - s_{i-1})$ for $i = 1, 2, \dots, m$. For their encoding we can use for example the δ-code [9, 4], which encodes a positive integer g in $1 + \lfloor \log g \rfloor + 2\lfloor \log(1 + \lfloor \log g \rfloor) \rfloor$ bits. Now, any consecutive $O(\log n)$ bits of S are decoded in constant time, using the fact that consecutive δ-codes stored in $O(\log n)$ bits can be decoded in constant time using table lookup.

We adopt another measure of entropy based on *gap encoding* to evaluate this method. In order to establish a matching lower bound on the space complexity, let

$\delta(g) = 1 + \lfloor \log g \rfloor$ be the minimum number of bits needed for representing $g \geq 1$. We define $gap(\hat{S})$ as in equation (1.2), and measure the size of the dictionary in terms of this *gap* function. We consider only the case $gap(\hat{S}) < n$ because otherwise we can use the data structure of Theorem 2.1.

THEOREM 2.2. *When $gap(\hat{S}) < n$, there exists a data structure for storing a set $\hat{S} \subset U = \{1, 2, \ldots, n\}$ in*

$$gap(\hat{S}) + \mathrm{O}(n \log\log n / \log n) \tag{2.7}$$

bits, such that rank and select can be supported in $\mathrm{O}(1)$ *time on the word RAM.*

Proof. We consider two cases, namely, (i) $m = \mathrm{O}(n/\log n)$ and (ii) $m = \Omega(n/\log n)$. In case (i), we use the data structure of Theorem 2.1. Since it is $gap(\hat{S}) = \mathrm{O}\big(m \log(n/m)\big) = \mathrm{O}(n \log\log n / \log n)$, the claim holds observing that $nH_k \leq \mathcal{B}(m, n) = \mathrm{O}(n \log\log n / \log n)$ as well. In case (ii), we use δ-codes for encoding $\hat{S}$ in $gap(\hat{S}) + \mathrm{O}(n \log\log n / \log n)$ bits. In order to decode these δ-codes, we divide the encoding into blocks of $\log n$ bits each, and store the position of the first δ-code in each block that does not overlap the previous block. These positions are encoded in

$$\mathrm{O}\left(\frac{gap(\hat{S})}{\log n} \log \frac{gap(\hat{S})}{gap(\hat{S})/\log n}\right) = \mathrm{O}\left(\frac{gap(\hat{S}) \log\log n}{\log n}\right)$$

bits by FID. This bound is $\mathrm{O}(n \log\log n / \log n)$ bits because $gap(\hat{S}) < n$. We also store the values s_i corresponding to the δ-codes whose positions are stored above by FID. Since $m = \Omega(n/\log n)$ implies $gap(\hat{S}) = \Omega(n/\log n)$, the size is

$$\mathrm{O}\left(\frac{gap(\hat{S})}{\log n} \log \frac{n}{gap(\hat{S})/\log n}\right) = \mathrm{O}\left(\frac{gap(\hat{S})}{\log n} \log \frac{n \log n}{n/\log n}\right)$$

which can be upper bounded as $\mathrm{O}(n \log\log n / \log n)$ bits, yielding equation (2.7).

3 Related Work and Concluding Remarks

We have presented a general approach for reducing the space complexity of succinct data structures to high-order empirical entropy bounds while preserving their asymptotical time complexity. We have made concrete examples of application in succinct dictionaries and succinct representation of trees. The idea of using the high-order empirical entropy, H_k, for analyzing the space complexity of data structures has been introduced by Ferragina and Manzini for the text indexing problem [11], using the the Burrow-Wheeler transform, and recently extended to labeled trees, with applications to XML [12]. Their methods, and the subsequent results by others, cannot be easily adapted to support the constant-time operation of decompressing a substring of $O(\log n)$ bits.

The high-order entropy data structuring approach has been confined to text algorithms until recently. Poon and Yiu [33] have proposed the first succinct data structure for range sum queries over a sequence S of n ω-bit integers with LZ78's entropy bounds, where $\omega = \mathrm{O}(\log n)$. The size of the data structure converges to $3nH$ bits where H is the entropy of S generated by a stationary ergodic information source. The query time is $\mathrm{O}\big(\frac{\log\log n}{\log\log\log n} + \omega\big)$ on the word RAM.

We designed the first suite of our data structures in Section 2.2, based on Poon and Yiu's idea of using LZ78. Therefore we describe our improvements. First, we improve the multiplicative constant in the data structure size from 3 to 1. This is achieved by the renumbering of phrases in array R, which can also be applied to other LZ78-based data structures. Second, we improve the query time from $\mathrm{O}\big(\frac{\log\log n}{\log\log\log n} + \omega\big)$ to constant time. Though they also propose a variation of the data structure whose size is $nH + \mathrm{o}(n)$ bits, the query time complexity increases to $\mathrm{O}(\log n)$. The most important improvement of ours is that we can decode consecutive $\mathrm{O}(\log n)$ bits of S in constant time using our novel idea of encoding edge labels of the LZ-trie and encoding dense regions separately. As previously mentioned, we obtain a general approach to improve previous bounds.

An independent and complementary approach to ours is that of Gupta, Hon, Shah and Vitter [20], who squeeze the space below the $\Omega(n \log\log n / \log n)$ bound of [24], so as to design succinct data structures that match the non-constant predecessor bound of [1]. Finally, the idea of getting the gap encoding bound of equation (1.2) for dictionaries has been proposed by Rajeev Raman at a Dagstuhl seminar in 2002.

Acknowledgments. The first author would like to thank Prof. C.K. Poon, who kindly provided [33]. The second author is in debt to Ankur Gupta, Rajeev Raman, and Jeff S. Vitter for many enlightening discussions on compressed data structures, and to Rajeev Raman for pointing out the attention to the gap bound.

References

[1] P. Beame and F. E. Fich. Optimal Bounds for the Predecessor Problem. In *Proc. ACM STOC*, pp. 295–304, 1999.

[2] M. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004.

[3] David Benoit, Erik D. Demaine, J. Ian Munro, and Venkatesh Raman. Representing trees of higher degree. In *Proc. WADS*, *LNCS* 1963, pp. 169–180, 1999.

[4] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A Locally Adaptive Data Compression Scheme. *Comm. of the ACM*, 29(4):320–330, April 1986.

[5] Daniel K. Blandford and Guy E. Blelloch. Compact representations of ordered sets. In *Proc. ACM-SIAM SODA*, pp. 11–19, 2004.

[6] Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. Compact representations of separable graphs. In *Proc. ACM-SIAM SODA*, pp. 679–688, 2003.

[7] Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM Journal on Computing*, 28(5):1627–1640, October 1999.

[8] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley-Interscience, 1991.

[9] P. Elias. Interval and Recency Rank Source Coding: Two On-Line Adaptive Variable-Length Schemes. *IEEE Trans. Inform. Theory*, IT-33(1):3–10, 1987.

[10] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Succinct Representation of Sequences. Tr/dcc-2004-5, August 2004. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/sequences.ps.gz`.

[11] Paolo Ferragina and Giovanni Manzini. On compressing and indexing data. *Journal of the ACM*, 2005.

[12] P. Ferragina, F. Luccio. G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. *Proc. IEEE FOCS*, 2005.

[13] Anna Gál and Peter Bro Miltersen. The cell probe complexity of succinct data structures. In *Proc. ICALP*, em LNCS 2719, pp. 332–344, 2003.

[14] R. F. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. In *Proc. CPM*, *LNCS* 3109, pp. 159–172, 2004.

[15] Richard F. Geary, Rajeev Raman, and Venkatesh Raman. Succinct ordinal trees with level-ancestor queries. In *Proc. ACM-SIAM SODA*, pp. 1–10, 2004.

[16] A. Golynski, J.I. Munro, and S.S. Rao. Rank/select operations on large alphabets: A tool for text indexing. In *Proc. ACM-SIAM SODA*, 2006, to appear.

[17] R. Grossi, A. Gupta, and J. S. Vitter. High-Order Entropy-Compressed Text Indexes. In *Proc. ACM-SIAM SODA*, pp. 841–850, 2003.

[18] R. Grossi, A. Gupta, and J. S. Vitter. When Indexing Equals Compression: Experiments with Compressing Suffix Arrays and Applications. In *Proc. ACM-SIAM SODA*, pp. 636–645, 2004.

[19] R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. In *Proc. ACM STOC*, pp. 397–406, 2000. Also, in *SIAM Journal on Computing*, 2005.

[20] A. Gupta, W.-K. Hon, R. Shah, and J. S. Vitter. Compressed Data Structures: Dictionaries and Data-Aware Measures. Manuscript, 2005.

[21] G. Jacobson. Space-efficient static trees and graphs. In *Proc. IEEE FOCS*, pp. 549–554, 1989.

[22] R. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 1999.

[23] M. Li and P. Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer Verlag, 1997.

[24] P. B. Miltersen. Lower bounds on the size of selection and rank indexes. In *Proc. ACM-SIAM SODA*, pp. 11–12, 2005.

[25] J. I. Munro. Tables. In *Proc. FSTTCS*, LNCS 1180, pp. 37–42, 1996.

[26] J. I. Munro and V. Raman. Succinct Representation of Balanced Parentheses and Static Trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[27] J. I. Munro, V. Raman, and S. S. Rao. Space Efficient Suffix Trees. *J. of Algorithms*, 39(2):205–222, 2001.

[28] J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct representations of permutations. In *Proc. ICALP*, *LNCS* 2719, pp. 345–356, 2003.

[29] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. on Computing*, 31(3):762–776, June 2002.

[30] J. Ian Munro, Venkatesh Raman, and Adam J. Storm. Representing dynamic binary trees succinctly. In *Proc. ACM-SIAM SODA*, pp. 529–536, 2001.

[31] J. Ian Munro and S. Srinivasa Rao. Succinct representations of functions. In *Proc. ICALP*, *LNCS* 3142, pp. 1006–1015, 2004.

[32] Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31:353–363, 2001.

[33] C. K. Poon and W. K. Yiu. Opportunistic Data Structures for Range Queries. In *Proc. COCOON*, LNCS 3595, pp. 560–569, 2005.

[34] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. ACM-SIAM SODA*, pp. 233–242, 2002.

[35] Rajeev Raman and S. Srinivasa Rao. Succinct dynamic dictionaries and trees. In *Proc. ICALP*, *LNCS* 2719, pp. 357–368, 2003.

[36] Kunihiko Sadakane. Succinct representations of *lcp* information and improvements in the compressed suffix arrays. In *Proc. ACM-SIAM SODA*, pp. 225–232, 2002.

[37] Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003.

[38] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inform. Theory*, IT-24(5):530–536, September 1978.