Succinct Indexes for Strings, Binary Relations and Multi-labeled Trees

Jérémy Barbay * Meng He * J. Ian Munro * S. Srinivasa Rao †

Abstract

We define and design succinct indexes for several abstract data types (ADTs). The concept is to design auxiliary data structures that occupy asymptotically less space than the information-theoretic lower bound on the space required to encode the given data, and support an extended set of operations using the basic operators defined in the ADT. As opposed to succinct (integrated data/index) encodings, the main advantage of succinct indexes is that we make assumptions only on the ADT through which the main data is accessed, rather than the way in which the data is encoded. This allows more freedom in the encoding of the main data. In this paper, we present succinct indexes for various data types, namely strings, binary relations and multi-labeled trees. Given the support for the interface of the ADTs of these data types, we can support various useful operations efficiently by constructing succinct indexes for them. When the operators in the ADTs are supported in constant time, our results are comparable to previous results, while allowing more flexibility in the encoding of the given data.

Using our techniques, we design a succinct encoding that represents a string of length n over an alphabet of size σ using $nH_k + o(n \lg \sigma)$ bits[1] to support access/rank/select operations in $o((\lg\lg\sigma)^3)$ time. We also design a succinct text index using $nH_k + o(n \lg \sigma)$ bits that supports pattern matching queries in $O(m \lg\lg\sigma + \mathsf{occ} \lg^{1+\epsilon} n \lg\lg\sigma)$ time, for a given pattern of length m. Previous results on these two problems either have a $\lg \sigma$ factor instead of $\lg\lg\sigma$ in terms of running time, or are not compressible.

1 Introduction

The rapid growth of large text sets and the need for efficient searches in these sets, has led to a trend of succinct representation of text indexes (perhaps including the text itself). *Succinct data structures* were first proposed by Jacobson [12] to encode bit vectors, (unlabeled) trees and planar graphs in space close to the information-theoretic lower bound, while supporting efficient navigational operations. This technique was successfully applied to various other abstract data types (ADTs), such as dictionaries, strings, binary relations [2] and labeled trees [2, 8]. In most of the previous results, researchers encode the given data, and use both the encoding and the auxiliary data structures to support various operations. Therefore, these techniques often require the given data to be stored in specific formats, e.g. [2, 7, 8, 10, 11]. We thus call this type of design *succinct encodings* of data structures.

The concept of *succinct indexes* was originally proposed to prove the lower bounds [5, 14] and to analyze the upper bounds [18] on the space required to encode some data structures: it limits the definition of the encoding to the index. In this paper, we formulate the distinction between the index and the raw data, and apply it to the design of succinct data structures. Given an ADT, our goal is to design auxiliary data structures (i.e. succinct indexes) that occupy asymptotically less space than the information-theoretic lower bound on the space required to encode the given data, and support an extended set of operations using the basic operators defined in the ADT. Succinct indexes and succinct encodings are closely related, but they are different concepts: succinct indexes make assumptions only on the ADT through which the given data is accessed, while succinct encodings represent data in specific formats. Succinct indexes are also more difficult to design: one can design a succinct encoding from a succinct index, but the converse is not true.

Although succinct indexes were previously presented primarily as a technical restriction to prove lower/upper bounds, we argue that in fact they are more adequate to the design of a library of succinct tools for multiple usages than succinct encodings, and that they are even directly required in certain applications. Some of the advantages of succinct indexes over succinct encodings are:

1. A succinct encoding requires the given data to be stored in a specific format. However, a succinct index applies to any encoding of the given data that supports a specific ADT. Thus when using succinct indexes, the given data can be either stored to

*Cheriton School of Computer Science, University of Waterloo, ON, Canada {jbarbay, mhe, imunro}@uwaterloo.ca

†Computational Logic and Algorithms group, IT University of Copenhagen, Copenhagen, Denmark ssrao@itu.dk

[1] We use $\lg n$ to denote $\lceil \log_2 n \rceil$.

achieve maximal compression or to achieve optimal support of the operations defined in the ADT.

2. The existence of two succinct encodings supporting two non-identical sets of operations over the same data type does not imply the existence of a single encoding supporting the union of the two sets of operations without storing the given data twice, because they may not store it in the same format. However, we can always combine two different succinct indexes for the same ADT to yield one index that supports the union of the two corresponding sets of operations in a straightforward manner.
3. In some cases, we do not need to store the given data explicitly because it can be computed from different but related data and still support the operations defined in the ADT. Hence a succinct index is the only additional memory cost.

In this paper, we design succinct indexes for strings, binary relations and multi-labeled trees on the standard word RAM model with word size $\Theta(\lg n)$, where n denotes the problem size. Given the support for the interface of these ADTs, we can support an extended set of operations efficiently. The succinct indexes occupy negligible space compared to the information-theoretic lower bound for representing the given data.

Based on the succinct index for strings, we design a succinct encoding that represents a string of length n over an alphabet of size σ using $nH_k + o(n \lg \sigma)$ bits[2], which supports access/rank/select operations in $o((\lg\lg\sigma)^3)$ time. We also design a succinct text index using $nH_k + o(n \lg \sigma)$ bits that supports searching for a pattern of length m in $O(m \lg\lg\sigma + \texttt{occ} \lg^{1+\epsilon} n \lg\lg\sigma)$ time (`occ` is the number of occurrences of the pattern).

2 Background

Here we outline the design of succinct data structures for several data types. We cite the results that we use in the design of succinct indexes, and those upon which we improve.

2.1 Bit Vectors A key structure we use is a bit vector B of size n that supports *rank* and *select* operations. We assume that the positions in B are numbered $1, 2, ..., n$. For $\alpha \in \{0, 1\}$, the operator $\texttt{bin\_rank}_B(\alpha, x)$ returns the number of occurrences of α in $B[1..x]$, and the operator $\texttt{bin\_select}_B(\alpha, r)$ returns the position of the r^{th} α in B. We omit the subscript B when it is clear from the context. Lemma 2.1 addresses the problem, in which part (a) is from Jacobson [12] and Clark and Munro [4], while part (b) is from Raman *et al.* [17].

[2] H_k denotes the k^th order entropy of a given string.

LEMMA 2.1. *A bit vector B of length n with v 1s can be represented using either: (a) $n + o(n)$ bits, or (b) $\lg \binom{n}{v} + O(n \lg\lg n / \lg n)$ bits, to support the access to each bit,* `bin_rank` *and* `bin_select` *in $O(1)$ time.*

A less powerful version of $\texttt{bin\_rank}(1, x)$, denoted $\texttt{bin\_rank}'(1, x)$, returns the number of 1s in $B[1..x]$ in the restricted case where $B[x] = 1$.

LEMMA 2.2. ([17]) *A bit vector B of length n with v 1s can be represented using $\lg \binom{n}{v} + o(v) + O(\lg\lg n)$ bits to support the access to each bit, $\texttt{bin\_rank}'(1, x)$ and $\texttt{bin\_select}(1, r)$ in $O(1)$ time.*

2.2 Strings and Binary Relations Grossi *et al.* [10] generalized `bin_rank` and `bin_select` operators to a string (or a sequence) S of length n over an alphabet of arbitrary size σ, and the operations include: $\texttt{string\_rank}(\alpha, x)$, which returns the number of occurrences of α in $S[1..x]$; $\texttt{string\_select}(\alpha, r)$, which returns the position of the r^{th} occurrence of α in the string; and $\texttt{string\_access}(x)$, which returns the character at position x in the string. They gave an encoding that takes $nH_0 + o(n \lg \sigma)$ bits to support these three operators in $O(\lg \sigma)$ time, where n is the length of the string. Golynski *et al.* [9] gave an encoding that uses $n(\lg \sigma + o(\lg \sigma))$ bits and supports $\texttt{string\_rank}(\alpha, x)$ and $\texttt{string\_access}(x)$ in $O(\lg\lg\sigma)$ time, and $\texttt{string\_select}(\alpha, r)$ in constant time.

Barbay *et al.* [2] extended the problem to the encoding of sequences of n objects where each object can be associated with a subset of labels from $[\sigma]$, this association being defined by a binary relation of t pairs from $[n] \times [\sigma]$. The operations include: $\texttt{label\_rank}(\alpha, x)$, which returns the number of objects labeled α up to (and including) x; $\texttt{label\_select}(\alpha, r)$, which returns the position of the r^{th} object labeled α; and $\texttt{label\_access}(x, \alpha)$, which checks whether object x is associated with label α. Their representation supports `label_rank` and `label_access` in $O(\lg\lg\sigma)$ time, and `label_select` in constant time using $t(\lg \sigma + o(\lg \sigma))$ bits[3].

2.3 Ordinal Trees An *ordinal tree* is a rooted tree in which the children of a node are ordered and specified by their ranks. Preorder and postorder traversals of such trees are well-known. We also use a different order for traversals, namely `DFUDS`, which is the order associated with the *depth first unary degree sequence* [3]

[3] In this paper, we assume that each object is associated with at least one label (thus $t \geq n$), and that $n \geq \sigma$. Barbay *et al.* [2] showed how to extend the results to other cases by simple reductions.

representation, where all the children of a node are listed before its other descendants (see Figure 2 in Section 3.3 for an example).

Various succinct data structures were designed to represent ordinal trees [3, 8, 12, 13, 16]. Benoit *et al.* [3] proposed the `DFUDS` representation of an ordinal tree using $2n+o(n)$ bits to support various navigational operations, which is close to the lower bound suggested by information theory ($2n - \Theta(\lg n)$ bits). Jansson *et al.* [13] extended this representation to support a richer set of navigational operations. Some of the operations supported in [3] and [13] are (we refer to each node by its preorder number):

- $\texttt{child}(x, i)$, i^{th} child of node x for $i \geq 1$;
- $\texttt{child\_rank}(x)$, number of left siblings of node x;
- $\texttt{depth}(x)$, depth of node x, i.e. the number of edges in the rooted path to x;
- $\texttt{level\_anc}(x, i)$, i^{th} ancestor of node x for $i \geq 0$ (given a node x at depth d, its i^{th} ancestor is the ancestor of x at depth $d - i$);
- $\texttt{nbdesc}(x)$, number of descendants of node x;
- $\texttt{degree}(x)$, degree of node x, i.e. the number of its children;
- $\texttt{LCA}(x, y)$, lowest common ancestor of x and y.

2.4 Labeled and Multi-Labeled Trees A *labeled tree* is a tree in which each node is associated with a label from a given alphabet $[\sigma]$, while in a *multi-labeled tree*, each node is associated with at least one label[4]. We use n to denote the number of nodes in a labeled / multi-labeled tree, and t to denote the total number of node-label pairs in a multi-labeled tree[5]. As we only consider ordinal trees, we assume that labeled / multi-labeled trees are ordinal trees.

Geary *et al.* [8] defined labeled extensions of the first six operators defined in Section 2.3. Their data structures support those in constant time, but use $2n + n(\lg\sigma + O(\sigma \lg\lg\lg n/\lg\lg n))$ bits, which is much more than the asymptotic lower bound of $n(\lg\sigma - o(\lg\sigma))$ suggested by information theory when σ is large. Ferragina *et al.* [6] proposed another structure for labeled trees that supports locating the first child of a given node x labeled α in constant time, and finding all the children of x labeled α in constant time per child. But it does not efficiently support the retrieval of the ancestors or descendants by labels. Also it uses $2n\lg\sigma + O(n)$ bits, which is almost twice the minimum space required to encode the tree. Barbay *et al.* [2] gave an encoding for labeled trees using $n(\lg\sigma + o(\lg\sigma))$ bits to support the retrieval of the ancestors or descendants by labels in $O(\lg\lg\sigma)$ time, which is generalized to represent multi-labeled trees in $t(\lg\sigma + o(\lg\sigma))$ bits.

[4]We use $[i]$ to denote the set $\{1, 2, ..., i\}$.

[5]In this paper, we assume that each node of the tree is associated with at least one label (thus $t \geq n$), and that $n \geq \sigma$. Barbay *et al.* [2] showed how to extend the results to other cases by simple reductions.

3 Succinct Indexes

We introduce succinct indexes in two steps: we first define the ADTs and then design succinct indexes for these ADTs.

3.1 Strings We first design succinct indexes for a given string S of length n over alphabet $[\sigma]$. We adopt the common assumption that $\sigma \leq n$ (otherwise, we can reduce the alphabet size to the number of characters that occur in the string). We define the ADT of a string through the `string_access` operator that returns the character at any given position of the string. To generalize the operators on strings defined in Section 2.2 to include "negative" searches, we define a *literal* as either a character, $\alpha \in [\sigma]$, or its negation:

Definition 3.1. *Consider a string $S[1..n]$ over the alphabet $[\sigma]$. A position $x \in [n]$* matches a literal *$\alpha \in [\sigma]$ if $S[x] = \alpha$. A position $x \in [n]$* matches the literal *$\bar{\alpha}$ if $S[x] \neq \alpha$. For simplicity, we define $[\bar{\sigma}]$ to be the set $\{\bar{1}, \ldots, \bar{\sigma}\}$.*

We then consider the following operators:

Definition 3.2. *Consider a string $S \in [\sigma]^n$, a literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ and a position $x \in [n]$ in S. The* α-predecessor *of position x, denoted by $\texttt{string\_pred}(\alpha, x)$, is the last position matching α before position x, if it exists. Similarly, the* α-successor *of position x, denoted by $\texttt{string\_succ}(\alpha, x)$, is the first position matching α after position x, if it exists.*

To illustrate the operations above, consider the string *bbaaacdd*. We have $\texttt{string\_pred}(a, 7) = 5$, as position 5 is the last position in the string before position 7 whose character is a. We also have $\texttt{string\_pred}(\overline{a}, 5) = 2$, as position 2 is the last position before position 5 whose character is not a. We use these definitions to state our results.

Lemma 3.1. *Given support for `string_access` in $f(n, \sigma)$ time on a string $S \in [\sigma]^n$, there is a succinct index using $n \cdot o(\lg\sigma)$ bits that supports `string_rank` for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ in $O(\lg\lg\sigma \lg\lg\lg\sigma(f(n, \sigma) + \lg\lg\sigma))$ time, and `string_select` for any character $\alpha \in [\sigma]$ in $O(\lg\lg\lg\sigma(f(n, \sigma) + \lg\lg\sigma))$ time.*

Proof. As $\texttt{string\_rank}(\overline{\alpha}, x) = x - \texttt{string\_rank}(\alpha, x)$ for $\alpha \in [\sigma]$, we only need to show how to support `string_rank` and `string_select` for $\alpha \in [\sigma]$.

We conceptually treat the given string S as an $n \times \sigma$ table E with rows indexed by $1, 2, ..., \sigma$ and columns by $1, 2, ..., n$. For any $\alpha \in [\sigma]$ and $x \in [n]$, entry $E[\alpha][x] = 1$ if $S[x] = \alpha$, and $E[\alpha][x] = 0$ otherwise. Reading E in row major order yields a conceptual bit vector A of length σn with exactly n 1s. We divide A into blocks of size σ. The *cardinality* of a block is the number of 1s in it. To make use of `string_access` to support operators on blocks, we group blocks into chunks. To be specific, we conceptually divide S into chunks of length σ (we assume that n is divisible by σ for simplicity), so that for the i^{th} chunk C, we have $C[j] = S[(i-1)\sigma + j]$, where $i \in [n/\sigma]$ and $j \in [\sigma]$. Thus each chunk consists of exactly σ blocks, one for each row of the chunk. We denote the block corresponding to the α^{th} row of a chunk C by C_α, where $\alpha \in [\sigma]$. We store the following data structures:

- For the entire string, we construct a bit vector B which stores the cardinalities of all the blocks in unary, in the order they appear in A, i.e. $B = 1^{l_1}01^{l_2}0...1^{l_n}0$, where l_i is the cardinality of the i^{th} block of A. The length of B is $2n$, as there are exactly n 1s in A, and n blocks. We store it using Part (a) of Lemma 2.1 in $2n + o(n)$ bits.
- For each chunk C, we construct a bit vector X that stores the cardinalities of the blocks in C in unary from top to bottom, i.e. $X_i = 01^{l_1}01^{l_2}0...1^{l_\sigma}0$, where l_α is the cardinality of block C_α. We store it in $2\sigma + o(\sigma)$ bits using Part (a) of Lemma 2.1 as its length is 2σ.
- For each chunk C, we construct an array R such that $R[j] = \texttt{bin\_rank}_D(1, j) \bmod \lg\sigma$, where D is the block $C_{C[j]}$. Each element of R is an integer in the range $[0, \lg\sigma - 1]$, so R can be stored in $\sigma \lg\lg\sigma$ bits.
- For each chunk C, we construct a conceptual permutation π on $[\sigma]$, defined later in the proof. We store an auxiliary structure P to support the access to π given π^{-1} using $O(\sigma \lg\sigma / \lg\lg\lg\sigma)$ bits [15].
- For each block C_α in a chunk C, let F_α be a conceptual, "sparsified" bit vector for C_α, in which only every $\lg\sigma^{\text{th}}$ 1 of C_α is present (i.e. $F_\alpha[j] = 1$ iff $C_\alpha[j] = 1$ and $\texttt{bin\_rank}(1, j)$ on C_α is divisible by $\lg\sigma$). We construct a y-fast trie [19] over F_α. This y-fast trie uses $O(\lg\sigma \times l_\alpha / \lg\sigma) = O(l_\alpha)$ bits (as the trie is on universe $[\sigma]$, we use a word size of $O(\lg\sigma)$ for it). The σ y-fast tries corresponding to all the blocks in a given chunk thus occupy $O(\sigma)$ bits in total.

Using the bit vector B, the support for `string_rank` and `string_select` operations on S can be reduced, in constant time, to supporting these operations on a given chunk (see Section 2 of [9]). Hence we only need to show how to support `string_rank` and `string_select` on a given chunk C.

We first show how to support $\texttt{bin\_rank}'(1, j)$ on block $D = C_{C[j]}$ (note that $D[j] = 1$). For this, we first compute $C[j]$ in $f(n, \sigma)$ time. Then we use the y-fast trie corresponding to $F_{C[j]}$ to compute $\lg\sigma\lfloor\texttt{bin\_rank}'_D(1, j)/\lg\sigma\rfloor$ in $O(\lg\lg\sigma)$ time, and retrieve $R[j]$ in constant time. The sum of the above two is the result. Thus $\texttt{bin\_rank}'_D(1, j)$ can be computed in $O(f(n, \sigma) + \lg\lg\sigma)$ time.

The permutation π for a chunk C can be obtained by writing down the positions (relative to the starting position of the chunk) of all the occurrences of each character α in increasing order, if α appears in C, for $\alpha = 1, 2, \cdots, \sigma$. Using π^{-1} to denote the inverse of π, we see that $\pi^{-1}(j)$ is equal to the sum of the following two values: the number of characters smaller than $C[j]$ in C, and $\texttt{bin\_rank}'(1, j)$ on block $D = C_{C[j]}$. The first value can be easily computed using X in constant time, and we have already shown how to compute the second value in $O(f(n, \sigma) + \lg\lg\sigma)$ time in the previous paragraph. Therefore, we can compute any element of π^{-1} in $O(f(n, \sigma) + \lg\lg\sigma)$ time. We can further use P to compute any element of π in $O(\lg\lg\lg\sigma(f(n, \sigma) + \lg\lg\sigma))$ time [15] (note that the $f(n, \sigma) + \lg\lg\sigma$ term in the above claim comes from the time required to retrieve a given element of π^{-1}).

Golynski *et al.* [9, Section 2.2] showed how to compute `string_select` on a chunk C by a single access to π plus a few constant-time operations. When combined with our approach, we can support `string_select` for any character $\alpha \in [\sigma]$ in $O(\lg\lg\lg\sigma(f(n, \sigma) + \lg\lg\sigma))$ time. They also showed how to compute `string_rank` by calling `string_select` $O(\lg\lg\sigma)$ times. Thus we can support operator `string_rank` in $O(\lg\lg\sigma \lg\lg\lg\sigma(f(n, \sigma) + \lg\lg\sigma))$ time.

As there are n/σ chunks, the sum of the space costs of the auxiliary structures constructed for all the chunks is clearly $O(n \lg\sigma / \lg\lg\lg\sigma)$ bits. The overall space cost of all the auxiliary structures is therefore $n \cdot o(\lg\sigma)$. □

LEMMA 3.2. *Using at most $2n + o(n)$ additional bits, the succinct index of Lemma 3.1 also supports* `string_pred` *and* `string_succ` *for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ in $O(\lg\lg\sigma \lg\lg\lg\sigma(f(n, \sigma) + \lg\lg\sigma))$ time.*

Proof. We show how to support `string_pred`; `string_succ` can be supported similarly. For $\alpha \in [\sigma]$, $\texttt{string\_pred}(\alpha, x) = \texttt{string\_select}(\alpha, \texttt{string\_rank}(\alpha, x) - 1)$. Hence we only need to show how to support $\texttt{string\_pred}(\alpha, x)$ when $\alpha \in [\bar{\sigma}]$.

We require another auxiliary structure. In the bit vector A, there are n 1s, so there are at most n runs of consecutive 1s. Assume that there are u runs and their lengths are $p_1, p_2, ..., p_u$, respectively. We store these lengths in unary using a bit vector U, i.e. $U = 1^{p_1}01^{p_2}0 \cdots 1^{p_u}0$. The length of U is $n + u \leq 2n$, and we store it using Part (a) of Lemma 2.1 in at most $2n + o(n)$ bits.

To support $\texttt{string\_pred}(\alpha, x)$ for $\alpha \in [\bar{\sigma}]$, let c be the character such that $\alpha = \bar{c}$. We first retrieve $S[x-1]$ in $f(n,\sigma)$ time. If $S[x-1] \neq c$, then we return $x-1$. Otherwise, we compute the number, j, of 1s up to position $(c-1)\sigma + x - 1$ in A (this position in A corresponds to the $(x-1)^{\text{th}}$ position in the c^{th} row in table E). Let C be the chunk that contains the $(n-1)^{\text{th}}$ position of S. As $j = \texttt{bin\_rank}_B(1, \texttt{bin\_select}_B(0, (c-1)n/\sigma + \lfloor (x-1)/\sigma \rfloor)) + \texttt{bin\_rank}'_{C_c}(1, (x-1) \bmod \sigma)$, we can compute j in $O(f(n,\sigma) + \lg\lg\sigma)$ time (the proof of Lemma 3.1 shows how to compute $\texttt{bin\_rank}'_D(1,k)$ in $O(f(n,\sigma) + \lg\lg\sigma)$ time, for any block D and position k such that $D[k] = 1$). The position in U that corresponds to the $(x-1)^{\text{th}}$ position in the c^{th} row in table E is $v = \texttt{bin\_select}_U(1, j)$. Thus the number of consecutive 1s preceding and including position v in U is $q = v - \texttt{bin\_select}_U(0, \texttt{bin\_rank}_U(0, v))$. If $q \geq x - 1$, then there is no 0 in front of position $x-1$ in row c of table E, so we return $-\infty$. Otherwise, we return $x - q - 1$ as the result. All the above operations take $O(f(n,\sigma) + \lg\lg\sigma)$ time. □

Combining Lemmas 3.1 and 3.2, we have our first main result:

Theorem 3.1. *Given support for* $\texttt{string\_access}$ *in* $f(n,\sigma)$ *time on a string* $S \in [\sigma]^n$, *there is a succinct index using* $n \cdot o(\lg\sigma)$ *bits that supports* $\texttt{string\_rank}$, $\texttt{string\_pred}$ *and* $\texttt{string\_succ}$ *for any literal* $\alpha \in [\sigma] \cup [\bar{\sigma}]$ *in* $O(\lg\lg\sigma \lg\lg\lg\sigma(f(n,\sigma) + \lg\lg\sigma))$ *time, and* $\texttt{string\_select}$ *for any character* $\alpha \in [\sigma]$ *in* $O(\lg\lg\lg\sigma(f(n,\sigma) + \lg\lg\sigma))$ *time.*

We can alternatively define the ADT of a string through the $\texttt{string\_select}(\alpha, r)$ operator, where $\alpha \in [\sigma]$. Although this definition seems unusual, it has a useful application in Section 4.2. With this definition, we have:

Corollary 3.1. *Given support for* $\texttt{string\_select}$ *(for any character* $\alpha \in [\sigma]$*) in* $f(n,\sigma)$ *time on a string* $S \in [\sigma]^n$, *there is a succinct index using* $n \cdot o(\lg\sigma)$ *bits that supports* $\texttt{string\_rank}$ *for any literal* $\alpha \in [\sigma] \cup [\bar{\sigma}]$ *and* $\texttt{string\_access}$ *in* $O(\lg\lg\sigma f(n,\sigma))$ *time.*

Proof. As in the proof of Theorem 3.1, we divide string S and its corresponding conceptual table E into chunks and blocks, and store bit vector B for the entire string and bit vector X for each chunk. We also store the same set of y-fast tries for chunks. With the $f(n,\sigma)$-time support for $\texttt{string\_select}$ on S, using the method described in the proof of Theorem 3.1, we can support $\texttt{string\_rank}$ on S in $O(\lg\lg\sigma)$ time.

Now we need to provide support for $\texttt{string\_access}$. We first design the data structures supporting the access to π and π^{-1} for any chunk C (see the proof of Theorem 3.1 for the definition of π and π^{-1}). We assume that C is the i^{th} chunk. From the definition of π we have that $\pi(j) = \texttt{bin\_select}_{C_\alpha}(1, r)$, where $\alpha = \texttt{bin\_rank}_X(0, \texttt{bin\_select}_X(1, j))$, and $r = \texttt{bin\_select}_X(1, j) - \texttt{bin\_select}_X(0, \alpha)$. α and r can clearly be computed in $O(1)$ time. As $\texttt{bin\_select}_{C_\alpha}(1, r) = \texttt{string\_select}(\alpha, r + z)$, where z is the number of 1s in the α^{th} row of E up to position $(i-1)\lg\sigma$ (we can compute z by performing rank/select operations on B in constant time), we can compute $\pi(j)$ in $f(n,\sigma)$ time. For each chunk C, instead of storing the auxiliary structure P in the proof of Theorem 3.1, we construct an auxiliary structure P' using $O(\sigma\lg\sigma/\lg\lg\sigma)$ bits to compute any element of π^{-1} in $O(\lg\lg\sigma f(n,\sigma))$ time [15]. Finally, we use the method of Golynski *et al.* [9, Section 2.2] to compute $C[j]$ in $O(\lg\lg\sigma f(n,\sigma))$ time using the access to π^{-1}, which in turn can be used to support $\texttt{string\_access}$ in $O(\lg\lg\sigma f(n,\sigma))$ time.

Similar to the proof of Theorem 3.1, the above auxiliary data structures (B, X, y-fast tries, and P') occupy $n \cdot o(\lg\sigma)$ bits. □

3.2 Binary Relations

We define the interface of the ADT of a binary relation through the following operator: $\texttt{object\_access}(x, i)$, which returns the i^{th} label associated with x in lexicographic order, and returns $+\infty$ if no such label exists. We also generalize the definition of literals to binary relations (i.e. the x^{th} object matches literal $\alpha \in [\sigma]$ if it is associated with label α; otherwise, it matches the literal $\bar{\alpha}$). We have:

Theorem 3.2. *Given support for* $\texttt{object\_access}$ *in* $f(n,\sigma,t)$ *time on a binary relation formed by* t *pairs from an object set* $[n]$ *and a label set* $[\sigma]$, *there is a succinct index using* $t \cdot o(\lg\sigma)$ *bits that supports* $\texttt{label\_rank}$ *for any literal* $\alpha \in [\sigma] \cup [\bar{\sigma}]$ *and* $\texttt{label\_access}$ *for any label* $\alpha \in [\sigma]$ *in* $O(\lg\lg\sigma \lg\lg\lg\sigma(f(n,\sigma,t) + \lg\lg\sigma))$ *time, and* $\texttt{label\_select}$ *for any label* $\alpha \in [\sigma]$ *in* $O(\lg\lg\lg\sigma(f(n,\sigma,t) + \lg\lg\sigma))$ *time.*

Proof. As with strings, we also conceptually treat a binary relation as an $n \times \sigma$ table E, and entry $E[\alpha][x] = 1$ iff object x is associated with label α. A binary relation on t pairs from $[n] \times [\sigma]$ can be stored as follows [2] (See Figure 1 for an example):

$$E=\begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

```
COLUMNS = 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1
ROWS    = 3, 4,    1, 4,    3,    1, 2, 3,    4
```

Figure 1: An example of the encoding of a binary relation.

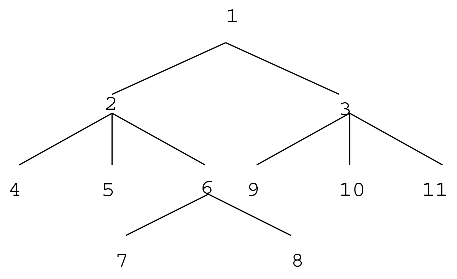
- a string ROWS of length t drawn from alphabet $[\sigma]$, such that the i^{th} label of ROWS is the label of the i^{th} pair in the column-major order traversal of E;
- a bit vector COLUMNS of length $n+t$ encoding the number of labels associated with each object in unary.

To design a succinct index for binary relations, we explicitly store the bit vector COLUMNS using Part (a) of Lemma 2.1 in $n+t+o(n+t)$ bits. We now show how to support string_access on ROWS using object_access. To compute the i^{th} character in ROWS, we need to compute the corresponding object, x, and the rank, r, of the corresponding label among all the labels associated with x. The position of the 0 in COLUMNS corresponding to the i^{th} character in ROWS is $l = \texttt{bin\_select}_{\texttt{COLUMNS}}(0, i)$. Therefore, $x = \texttt{bin\_rank}_{\texttt{COLUMNS}}(1, l) + 1$, and $r = l - \texttt{bin\_select}_{\texttt{COLUMNS}}(1, x-1)$ if $x > 1$ ($r = l$ otherwise). Thus with these additional operations, we can support string_access using one call to object_access in addition to some constant-time operations.

We store a succinct index for ROWS using Theorem 3.1 in $t \cdot o(\lg \sigma)$ bits. As we can support string_access on ROWS using object_access, the index can support string_rank and string_select on ROWS for any label $\alpha \in [\sigma]$. Using the approach of Barbay *et al.* [2, Theorem 1] to support label_rank, label_select and label_access operations on binary relations using rank/select on ROWS and COLUMNS, we can support these operators. The run times of the algorithms can be easily computed from Theorem 3.1 and [2]. The space of the index is the sum of space cost of storing COLUMNS and the index for ROWS, which is $t \cdot o(\lg \sigma)$. □

3.3 Multi-Labeled Trees We define the interface of the ADT of a multi-labeled tree through the following operator: $\texttt{node\_label}(x, i)$, which returns the i^{th} label associated with node x in lexicographic order. We store the tree structure as part of the index (as it takes negligible space), and hence do not assume the support for any navigational operation in the ADT. Recall that we refer to nodes by their preorder numbers (i.e. node x is the x^{th} node in the preorder traversal).

To support the navigational operations on an ordinal tree, we have the following lemma:



DFUDS: ((() ((())) (())) ((())))

Figure 2: An ordinal tree (where each node is assigned its rank in DFUDS order) and its DFUDS representation [3].

Lemma 3.3. *Using the* DFUDS *representation [3, 13], an ordinal tree with n nodes can be encoded in $2n + o(n)$ bits to support all the navigational operations defined in Section 2.3 and the following operations in $O(1)$ time:*

- $\texttt{find\_dfuds}(i)$*, the rank in* DFUDS *order of the i^{th} node in preorder;*
- $\texttt{find\_pre}(i)$*, the rank in preorder of the i^{th} node in* DFUDS *order.*

Proof. As it is shown in [3] and [13] how to support all the navigational operations defined in Section 2.3, we only need to provide support for $\texttt{find\_dfuds}(i)$ and $\texttt{find\_pre}(i)$. In the rest of the proof, we use the operators defined in [3], which are supported in constant time.

In the balanced parentheses representation of the DFUDS sequence of the tree [3], each node corresponds to an opening parenthesis and a closing parenthesis. We observe that in the sequence, the opening parentheses correspond to DFUDS order, while the closing parentheses correspond to the preorder. For example, in Figure 2, the 6th node in DFUDS order (which is the 5th node in preorder) corresponds to the 6th opening parenthesis, and the 5th closing parenthesis.

With this observation, $\texttt{find\_dfuds}(i)$ means that for the node, x, that corresponds to the i^{th} closing parenthesis, we need to compute the rank of the corresponding opening parenthesis among opening parentheses. To compute this value, we first find the opening parenthesis that matches the closing parenthesis that comes before node x. Its position in the sequence is: $j = \texttt{find\_open}(\texttt{select}_{\texttt{close}}(i-1))$. With j, we can compute the position of the parent of x, which is $p = \texttt{select}_{\texttt{close}}(\texttt{rank}_{\texttt{close}}(j)) + 1$, and $\texttt{child\_rank}(x)$ (denoted by r), which is $r = \texttt{select}_{\texttt{close}}(\texttt{rank}_{\texttt{close}}(p) + 1) - j$. Finally, $\texttt{rank}_{\texttt{open}}(p + r - 1)$ is the result.

The computation of $\texttt{find\_pre}(i)$ is exactly the inverse of the above process. $\square$

We now define permuted binary relations and present a related lemma that we need to design succinct indexes for multi-labeled trees.

DEFINITION 3.3. *Given a permutation π on $[n]$ and a binary relation $R \subset [n] \times [\sigma]$, the* permuted binary relation $\pi(R)$ *is the relation such that $(x, \alpha) \in \pi(R)$ if and only if $(\pi^{-1}(x), \alpha) \in R$.*

LEMMA 3.4. *Consider a permutation π on $[n]$, such that the access to $\pi(i)$ and $\pi^{-1}(i)$ is supported in $O(1)$ time. Given a binary relation $R \subset [n] \times [\sigma]$ of cardinality t, and support for* `object_access` *on R in $f(n, \sigma, t)$ time, there is a succinct index using $t \cdot o(\lg \sigma)$ bits that supports* `label_rank` *and* `label_access` *in $O(\lg\lg\sigma \lg\lg\lg\sigma(f(n, \sigma, t) + \lg\lg\sigma))$ time, and* `label_select` *in $O(\lg\lg\lg\sigma(f(n, \sigma, t) + \lg\lg\sigma))$ time, on* both *R and $\pi(R)$ for any label $\alpha \in [\sigma]$.*

Proof. The proof of Theorem 3.2 shows how to support `string_access` on `ROWS` using `object_access` on R and `bin_rank`/`bin_select` on `COLUMNS`, which allows us to design a succinct index for R using a succinct index for `ROWS` and the bit vector `COLUMNS`. We denote `ROWS`$'$ and `COLUMNS`$'$ to be the string and bit vector corresponding to $\pi(R)$, and store `COLUMNS`$'$ in $n+t+o(n+t)$ bits using using Part (a) of Lemma 2.1. Thus to design a succinct index to support efficient retrieval for both R and $\pi(R)$, we only need to show how to support `string_access` on the string `ROWS`$'$.

To support $\texttt{string\_access}(i)$ on `ROWS`$'$, we first compute the object x corresponding to the i^{th} element of `ROWS`$'$ using $x = \texttt{bin\_rank}_{\texttt{COLUMNS}'}(1, \texttt{bin\_select}_{\texttt{COLUMNS}'}(0, i)) + 1$. Let $r = i - \texttt{bin\_select}_{\texttt{COLUMNS}'}(1, x - 1)$. We have that the i^{th} element of `ROWS`$'$ corresponds to the r^{th} label of x in $\pi(R)$. As object x corresponds to object $y = \pi^{-1}(x)$ in `ROWS`, we have that $\texttt{string\_access}_{\texttt{ROWS}'}(i) = \texttt{object\_access}_R(y, r)$. Thus we can support $\texttt{string\_access}_{\texttt{ROWS}'}(i)$ in $f(n, \sigma, t)$ time. $\square$

To efficiently find all the α-ancestors of any given node, for each node and for each of its labels α we encode the number of α-ancestors of x. To measure the maximum number of such ancestors, we define the *recursivity* of a node, motivated by the notion of *document recursion level* of a given XML document [20].

DEFINITION 3.4. *The* recursivity ρ_α *of a label α in a multi-labeled tree is the maximum number of occurrences of α on any rooted path of the tree. The* average recursivity ρ *of a multi-labeled tree is the average recursivity of the labels weighted by the number of nodes associated with each label α (denoted by t_α): $\rho = \frac{1}{t}\sum_{\alpha \in [\sigma]}(t_\alpha \rho_\alpha)$.*

Note that ρ is usually small in practice, especially for XML trees. Zhang *et al.* [20] observed that in practice the document recursion level (when translated to our more precise definition, it is the maximum value of all ρ_α's minus one, which can be easily used to bound ρ) is often very small: in their data sets, it was never larger than 10. With this definition, we have:

THEOREM 3.3. *Consider a multi-labeled tree on n nodes and σ labels, associated in t relations, of average recursivity ρ. Given support for* `node_label` *in $f(n, \sigma, t)$ time, there is a succinct index using $t \cdot o(\lg \sigma)$ bits that supports (for a given node x) the enumeration of:*

- *the set of α-descendants of x (denoted by D) in $O(|D| \lg\lg\sigma \lg\lg\lg\sigma(f(n, \sigma, t) + \lg\lg\sigma))$ time;*
- *the set of α-children of x (denoted by C) in $O(|C| \lg\lg\sigma \lg\lg\lg\sigma(f(n, \sigma, t) + \lg\lg\sigma))$ time;*
- *the set of α-ancestors of x (denoted by A) in $O(\lg\lg\sigma \lg\lg\lg\sigma(f(n, \sigma, t) + \lg\lg\sigma) + |A|(\lg\lg\rho_\alpha + \lg\lg\lg\sigma(f(n, \sigma, t) + \lg\lg\sigma)))$ time using $t(\lg\rho + o(\lg\rho))$ bits of extra space.*

Proof. We encode the underlying ordinal tree structure in $2n + o(n)$ bits using Lemma 3.3. The sequence of nodes referred by their preorder (`DFUDS` order) numbers and the associated label sets form a binary relation R_p (R_d). Lemma 3.3 provides constant-time conversions between the preorder numbers and the `DFUDS` order numbers, and `node_label` supports `object_access` on R_p. By Lemma 3.4, we can construct succinct indexes for R_p and R_d using $t \cdot o(\lg \sigma)$ bits, and support `label_rank`, `label_select` and `label_access` operations on either of them efficiently.

Using the technique of Barbay *et al.* [2, Corollary 1], the succinct index for R_p enables us to enumerate all the descendants of a node x matching label α in $O(|D| \lg\lg\sigma \lg\lg\lg\sigma(f(n, \sigma, t) + \lg\lg\sigma))$ time (we can alternatively use the succinct index for R_d to achieve the same result). Similarly, the succinct index of R_d enables us to enumerate all children of a node x matching α in $O(|C| \lg\lg\sigma \lg\lg\lg\sigma(f(n, \sigma, t) + \lg\lg\sigma))$ time, as the `DFUDS` order traversal lists the children of any given node consecutively.

As there is no order in which the ancestors of each node are consecutive, we store for each label α of a node x the number of ancestors of x matching α.

To be specific, for each label α such that $\rho_\alpha > 1$, we represent those numbers in one string $S_\alpha \in [\rho_\alpha]^{t_\alpha}$ (see Definition 3.4), where the i^{th} number of S_α corresponds to the i^{th} node labeled α in preorder. As the lengths of the strings $(S_\alpha)_{\alpha\in[\sigma]}$ are implicitly encoded in R_p, we encode for each label α its recursivity ρ_α in unary, using at most $t + \sigma + o(t + \sigma)$ bits. We use the encoding of Golynski *et al.* [9, Theorem 2.2] to encode each string S_α in $t_\alpha(\lg \rho_\alpha + o(\lg \rho_\alpha))$ bits to support `string_rank` and `string_access` in $O(\lg\lg \rho_\alpha)$ time and `string_select` in constant time. The total space used by these strings is $\sum_{\alpha\in[\sigma]} t_\alpha(\lg \rho_\alpha + o(\lg \rho_\alpha))$. By concavity of the logarithmic function, this is at most $\left(\sum_{\alpha\in[\sigma]} t_\alpha\right)\left(\lg\left(\frac{\sum_{\alpha\in[\sigma]} t_\alpha \rho_\alpha}{\sum_{\alpha\in[\sigma]} t_\alpha}\right) + o\left(\frac{\sum_{\alpha\in[\sigma]} t_\alpha \rho_\alpha}{\sum_{\alpha\in[\sigma]} t_\alpha}\right)\right) = t(\lg \rho + o(\lg \rho))$.

To support the enumeration of all the α-ancestors of a node x, we first find from R_p the number, p_x, of α-nodes preceding x in preorder using `label_rank`. Then we iterate i from 1. In each iteration, we first find the position p_i in S_α of the character i immediately preceding position p_x: it corresponds to the $p_i{}^{\text{th}}$ α-node in preorder (this can be located using `label_select` on R_p). If this node is an ancestor of x (this can be checked using `depth` and `level_anc` in constant time), output it, increment i and iterate, otherwise stop. Each iteration contains a `label_select` on R_p and some rank and select operations on S_α, so each is performed in $O(\lg\lg \rho_\alpha + \lg\lg\lg \sigma(f(n,\sigma,t) + \lg\lg \sigma))$ time. Hence it takes $O(\lg\lg \sigma \lg\lg\lg \sigma(f(n,\sigma,t) + \lg\lg \sigma) + |A|(\lg\lg \rho_\alpha + \lg\lg\lg \sigma(f(n,\sigma,t) + \lg\lg \sigma)))$ time to enumerate A. □

We can also support the retrieval of the first α-descendant, child or ancestor of node x that appears after node y in preorder:

COROLLARY 3.2. *The structure above also supports (for any two given nodes x and y) the selection of:*

- *the first α-descendant of x after y in preorder in $O(\lg\lg \sigma \lg\lg\lg \sigma(f(n,\sigma,t) + \lg\lg \sigma))$ time;*
- *the first α-child of x after y in preorder in $O(\lg\lg \sigma \lg\lg\lg \sigma(f(n,\sigma,t) + \lg\lg \sigma))$ time;*
- *the first α-ancestor of x after y in preorder in $O(\lg\lg \rho_\alpha + \lg\lg \sigma \lg\lg\lg \sigma(f(n,\sigma,t) + \lg\lg \sigma))$ time.*

Proof. Using the index in Theorem 3.3, we can easily support the first operation. The support for the second operation is nontrivial only when y is a descendant of x. In this case, we first locate the child of x, node u, that is also an ancestor of y using `depth` and `level_anc`. Then the problem is reduced to the selection of the first α-child of x after u in preorder, which can be computed by performing rank/select operations on R_d.

To support the search for the first α-ancestor of x after y, we assume that y precedes x in preorder (otherwise the operator returns ∞), and that y is an ancestor of x (if not, the problem can be reduced to the search for the first α-ancestor of node x after node $\texttt{LCA}(x, y)$). Using rank and select on the relation R_p and some navigational operators, we can find the first α-descendant z of y in preorder in $O(\lg\lg \sigma \lg\lg\lg \sigma(f(n,\sigma,t) + \lg\lg \sigma))$ time. Node z is not necessarily an ancestor of x, but it has the same number, i, of α-ancestors as the node we are looking for. We can retrieve i from the string S_α in $O(\lg\lg \rho_\alpha)$ time. Finally, the first α-ancestor of x after y is the α-node corresponding to the value i immediately preceding the position corresponding to x in S_α, found in $O(\lg\lg \sigma \lg\lg\lg \sigma(f(n,\sigma,t) + \lg\lg \sigma) + \lg\lg \rho_\alpha)$ time. □

The operations on multi-labeled trees are important for the support of XPath queries for XML trees [1, 2]. The main idea of our algorithms is to construct indexes for binary relations for different traversal orders of the trees. Note that without succinct indexes, we would need to encode different binary relations separately and waste a lot of space.

4 Applications

4.1 High-Order Entropy-Compressed Succinct Encodings for Strings

Given a string S of length n over alphabet $[\sigma]$, we now design a high-order entropy-compressed succinct encoding for it that supports `string_access`, `string_rank`, and `string_select` efficiently. Golynski *et al.* [9] considered the problem and suggested a method with space requirements propotional to the k^{th} order entropy of a different but related string. Here we solve the problem in its original form.

THEOREM 4.1. *A string S of length n over alphabet $[\sigma]$ can be represented using $nH_k + o(n \lg \sigma)$ bits, for any positive integer k such that $k + \lg \sigma = o(\lg n)$, to support* `string_access` *in $O(1)$ time. The representation also supports* `string_rank`*,* `string_pred` *and* `string_succ` *for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ in $O((\lg\lg \sigma)^2 \lg\lg\lg \sigma)$ time, and* `string_select` *for any character $\alpha \in [\sigma]$ $O(\lg\lg \sigma \lg\lg\lg \sigma))$ time.*

Proof. We use the approach presented by Sadakane and Grossi [18] to store S in $nH_k + O(n \lg \sigma \lg\lg n / \lg n)$ (H_k denotes the k^{th} order entropy of S) for any positive integer k such that $k + \lg \sigma = o(\lg n)$. This representation allows us to retrieve $S[i]$ in $O(1)$ time (i.e. operator `string_access` can be supported in $O(1)$ time). We store a succinct index for S using Theorem 3.1, and the support for the above operations immediately follows. The overall space is $nH_k + O(n \lg \sigma \lg\lg n / \lg n) +$

$O(n \lg\sigma/\lg\lg\lg\sigma)$. The last two terms sum up to $O(n \lg\sigma(\lg\lg n/\lg n + 1/\lg\lg\lg\sigma)) = o(n\lg\sigma)$.[6] □

Using similar approaches, we can design succinct encodings for binary relations and multi-labeled trees based on our succinct indexes, and compress the underlying strings (recall that we reduce the operations on binary relations and multi-labeled trees to rank/select on strings and bit vectors) to high-order entropies.

4.2 High-Order Entropy-Compressed Text Indexes for Large Alphabets

Text indexes are data structures that facilitate text searching. Given a text T of length n and a pattern P of length m, whose symbols are drawn from the same fixed alphabet Σ, the goal is to look for the occurrences of P in T. We consider three types of queries: existential queries, cardinality queries, and listing queries. An *existential query* returns a boolean value that indicates whether P occurs as a substring in T. A *cardinality query* returns the number, `occ`, of occurrences of P in T. A *listing query* lists all the positions of occurrences of P in T.

We now apply our index to design space-efficient suffix arrays. We first present the following lemma to encode strings in 0th order entropy while supporting rank and select:

LEMMA 4.1. *A string S of length n over alphabet $[\sigma]$ can be represented using $n(H_0 + o(\lg\sigma))$ bits to support* `string_access` *and* `string_rank` *for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ in $O(\lg\lg\sigma)$ time, and* `string_select` *for any character $\alpha \in [\sigma]$ in $O(1)$ time.*

Proof. As in the proof of Theorem 3.1, we consider the conceptual table E for string S. Each row of E is a bit vector, and we denote the α^{th} row by $E[\alpha]$ for $\alpha \in [\sigma]$. For each $\alpha \in [\sigma]$, we store $E[\alpha]$ using Lemma 2.2 in $\lg\binom{n}{n_\alpha} + o(n_\alpha) + O(\lg\lg n) \approx n_\alpha \lg\frac{en}{n_\alpha} + o(n_\alpha) + O(\lg\lg n)$ bits, where n_α is the number of occurrences of α in S. Summing the space cost of all the $E[\alpha]$'s for $\alpha \in [\sigma]$, the last two terms clearly sum to $n \cdot o(\lg\sigma)$, while the first term on the right-hand side sums to $nH_0 + n\lg e$. Therefore, the total space cost is $n(H_0 + o(\lg\sigma))$ bits.

With E stored as above, `string_select` can be supported in $O(1)$ time, as `string_select`(α, i) = `bin_select`$_{E[\alpha]}(1, i)$, for $\alpha \in [\sigma]$. With the constant-time support for `string_select` on S, we can construct a succinct index using Corollary 3.1 to support `string_rank` and `string_access` in $O(\lg\lg n)$ time. This index uses $n \cdot o(\lg\sigma)$ bits, so the overall space cost is $n(H_0 + o(\lg\sigma))$ bits. □

[6] If $\sigma < \sqrt{\lg n}/2$, we can support all the operations in constant time using table lookups. When $\sigma \geq \sqrt{\lg n}/2$, we can bound n in terms of σ, and hence this term is $o(n\lg\sigma)$.

We can represent suffix arrays by encoding the Burrows-Wheeler transformed string of the raw text (denoted by $T^{\texttt{BWT}}$) appropriately [7, 11]. Ferragina *et al.* [7] also presented how to design a high-order entropy-compressed suffix array given an encoding of $T^{\texttt{BWT}}$ that occupies space in 0th order entropy plus an appropriate lower order term. Combining these results with Lemma 4.1 yields:

THEOREM 4.2. *A text string T of length n over alphabet $[\sigma]$ can be stored using $nH_k + o(n\lg\sigma))$ bits for any $k \leq \beta\lg_\sigma n$ and $0 < \beta < 1$. Given a pattern P of length m, this encoding can answer existential and cardinality queries in $O(m\lg\lg\sigma)$ time, list each occurrence in $O(\lg^{1+\epsilon} n\lg\lg\sigma)$ time for any ϵ where $0 < \epsilon < 1$, and output a substring of length l in $O((l + \lg^{1+\epsilon} n)\lg\lg\sigma)$ time.*

Grossi *et al.* [10] designed a text index that uses $nH_k + o(n\lg\sigma)$ bits, and supports existential and cardinality queries in $O(m\lg\sigma + \texttt{polylog}(n))$ time. Golynski *et al.* [9] reduced the $\lg\sigma$ factor in the query time to a $\lg\lg\sigma$, but their index is not compressible. Our text index has the advantages of both these indexes.

5 Conclusions

In this paper, we define succinct indexes for the design of data structures. We show their advantages (listed in Section 1) by presenting succinct indexes for strings, binary relations and multi-labeled trees, and applying them to various applications.

The concept of succinct indexes is of both theoretical and practical importance to the design of data structures. In theory, the separation of the ADT and the index enables researchers to design an encoding of the given data to achieve desired results or trade-offs more easily, as the encoding only needs to support the ADT. In addition, to support new operations, researchers merely need to design additional succinct indexes without redesigning the whole structure. In practice, this concept allows developers to engineer the implementation of ADTs and succinct indexes separately, and the fact that multiple succinct indexes for the same ADT can be easily combined to provide one succinct index makes it possible to further divide the implementation of succinct indexes into several (possibly concurrent) steps. This is good software engineering practice, to allow separated testing and concurrent development, and to facilitate the design of expandable software libraries. Furthermore, succinct indexes provide a way to support efficient operations on implicit data, which is common in both theory and practice. We thus expect that the concept of succinct indexes will influence the design of succinct data structures.

References

[1] Jérémy Barbay. Adaptive search algorithm for patterns, in succinctly encoded XML. Technical Report CS-2006-11, University of Waterloo, Ontario, Canada, 2006.

[2] Jérémy Barbay, Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching*, pages 24–35. Springer-Verlag LNCS 4009, 2006.

[3] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[4] David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391, 1996.

[5] Erik D. Demaine and Alejandro Lopez-Ortiz. A linear lower bound on index size for text retrieval. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 289–294, 2001.

[6] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proceedings of the 46th IEEE Symposium on Foundations of Computer Science*, pages 184–196, 2005.

[7] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. An alphabet-friendly FM-index. In *Proceedings of the 11th Symposium on String Processing and Information Retrieval*, pages 150–160. Springer-Verlag LNCS 3246, 2004.

[8] Richard F. Geary, Rajeev Raman, and Venkatesh Raman. Succinct ordinal trees with level-ancestor queries. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1–10, 2004.

[9] Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 368–373, 2006.

[10] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 841–850, 2003.

[11] Meng He, J. Ian Munro, and S. Srinivasa Rao. A categorization theorem on suffix arrays with applications to space efficient text indexes. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 23–32, 2005.

[12] Guy Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 549–554, 1989.

[13] Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Ultra-succinct representation of ordered trees. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2007.

[14] Peter Bro Miltersen. Lower bounds on the size of selection and rank indexes. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 11–12, 2005.

[15] J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct representations of permutations. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*, pages 345–356, 2003.

[16] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[17] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 233–242, 2002.

[18] Kunihiko Sadakane and Roberto Grossi. Squeezing succinct data structures into entropy bounds. In *Proceedings of the 17th annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1230–1239, 2006.

[19] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983.

[20] Ning Zhang, M. Tamer Özsu, Ashraf Aboulnaga, and Ihab F. Ilyas. XSEED: Accurate and Fast Cardinality Estimation for XPath Queries. In *Proceedings of the 22nd International Conference on Data Engineering*, pages 61–72, 2006.