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

JÉRÉMY BARBAY, University of Chile
MENG HE and J. IAN MUNRO, University of Waterloo
SRINIVASA RAO SATTI, Seoul National University

We define and design succinct indexes for several abstract data types (ADTs). The concept is to design auxiliary data structures that ideally 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. The main advantage of succinct indexes as opposed to succinct (integrated data/index) encodings 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 $nH_k(S)+\lg\sigma\cdot o(n)+O(\frac{n\lg\sigma}{\lg\lg\lg\sigma})$ bits to support access/rank/select operations in $o((\lg\lg\sigma)^{1+\epsilon})$ time, for any fixed constant $\epsilon>0$. We also design a succinct text index using $nH_0(S)+O(\frac{n\lg\sigma}{\lg\lg\sigma})$ bits that supports finding all the $\mathtt{occ}$ occurrences of a given pattern of length m in $O(m\lg\lg\sigma+\mathtt{occ}\lg n/\lg^{\epsilon}\sigma)$ time, for any fixed constant $\epsilon>0$. Previous results on these two problems either have a $\lg\sigma$ factor instead of $\lg\lg\sigma$ in the running time, or are not compressed. Finally, we present succinct encodings of binary relations and multi-labeled trees that are more compact than previous structures.

Categories and Subject Descriptors: E.1 [**Data**]: Data Structures—*Arrays*; *Trees*; E.4 [**Data**]: Coding and Information Theory—*Data compaction and compression*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Computations on discrete structures*; E.2 [**Data**]: Data Storage Representations; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Indexing methods*

General Terms: Algorithms, Design, Theory

The preliminary version of this paper was published in the *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 2007, 680-689.
This work was supported by NSERC of Canada and the Canada Research Chairs program. The work was done when the first author was in Cheriton School of Computer Science, University of Waterloo, Canada, and the fourth author was in Computational Logic and Algorithms group, IT University of Copenhagen, Denmark.
Authors' addresses: Jérémy Barbay, Department of Computer Science (DCC), University of Chile, Avenida Blanco Encalada 2120, Tercer Piso, Santiago, C.P. 837-0459, Chile, email: jeremy.barbay@dcc.uchile.cl; Meng He, J. Ian Munro, Cheriton School of Computer Science, University of Waterloo, 200 University Avenue West, Waterloo, Ontario, N2L 3G1, Canada, email: {mhe, imunro}@uwaterloo.ca; Srinivasa Rao Satti, School of Computer Science and Engineering, Seoul National University, 599 Gwanakro, Gwanak-Gu, Seoul 151-744, South Korea, email: ssrao@cse.snu.ac.kr.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

Additional Key Words and Phrases: succinct data structures, succinct indexes, strings, binary relations, trees, labeled trees, multi-labeled Trees, text indexing, compressed text indexes

ACM Reference Format:
Barbay, J., He, M., Munro, J. I., and Satti, S. S. 2011. Succinct Indexes for Strings, Binary Relations and Multi-labeled Trees. ACM Trans. Algor. V, N, Article A (January YYYY), 27 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

The rapid growth of large sets of text and the need for efficient searches of these sets, have led to a trend of succinct representation of text indexes as well as the text itself. ***Succinct data structures*** were first proposed by Jacobson [1989] 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 [Raman et al. 2007], strings [Grossi et al. 2003; Barbay et al. 2007], binary relations [Barbay et al. 2007] and labeled trees [Geary et al. 2006; Ferragina et al. 2005; Barbay et al. 2007]. In addition, succinct data structures have been proved to be very useful in practice. For example, Delpratt *et al.* [2006] engineered the implementation of succinct trees and reported that their structure uses 3.12 to 3.81 bits per node to encode the structures of XML trees that have 57K to 160M nodes. Such space cost is merely a small percentage of that of an explicit, pointer-based tree representation.

In most of the previous work, researchers encode the given data (or assume that the data is encoded) in a specific format, and further construct auxiliary data structures on it. They then use both the encoded data and the auxiliary data structures to support various operations, e.g. [Grossi et al. 2003; Geary et al. 2006; Ferragina et al. 2004; He et al. 2005; Barbay et al. 2007]. Usually in this type of design, the auxiliary data structures do not work if the given data is encoded in a different format, and therefore, the encoding of the given data and the design of the auxiliary data structures are inseparable. We thus call this type of design ***succinct integrated encoding*** of data structures.

A different line of research concentrates on reducing the size of the traditional text indexes to allow fast text retrieval, without transforming the text (i.e. the given data) to store it in specific formats. Therefore, in such research work, the representation of the text indexes and the encodings of the text itself can be designed separately. For example, Clark and Munro [1996] designed a compact PAT tree which takes much less space than the standard representation of a suffix tree, and used it to facilitate text retrieval.

The concept of separating the index and the given data was also used to prove the lower bounds [Demaine and López-Ortiz 2003; Gál and Miltersen 2003; Miltersen 2005; Golynski 2007] and to analyze the upper bounds [Sadakane and Grossi 2006] on the space required to encode some data structures. For example, Demaine and López-Ortiz [2003] proved that any text index supporting pattern search in time linear in the length of the pattern requires roughly the same amount of space as the text itself. Miltersen [2005] proved a lower bound of the size of any index supporting rank/select operations on bit vectors, and Golynski [2007] further improved his results. Sadakane and Grossi [2006] analyzed the space cost of their data structure by proving that the auxiliary data structures occupy asymptotically less space than the given data.

In this paper, we formalize 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 ideally 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 de-

fined in the ADT. Although the key idea here is still separating the index and the given data, our model differs from previous models. The ***index model*** or ***systematic data structures*** used in previous work on proving lower bounds [Demaine and López-Ortiz 2003; Gál and Miltersen 2003; Miltersen 2005; Golynski 2007] and on reducing the size of text indexes [Clark and Munro 1996] require the raw data to be kept "verbatim" (for example, a string must be stored physically as an array of characters in computer memory), while our model does not make any assumptions on the format. The work of Sadakane and Grossi [2006] focused on designing a compact storage scheme of data that support access operators, while we aim at supporting a richer set of operations, and the access operators are often used as the basic operators defined in the ADT in our model.

Succinct indexes and succinct integrated 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 integrated encodings represent data in specific formats. Succinct indexes are also more difficult to design: raw data plus a succinct index is a succinct integrated encoding, but it might not be possible to split a succinct integrated encoding into raw data and a succinct index.

Although the concept of succinct indexes was previously adopted mainly to design space efficient text indexes, and was also presented as a technical restriction to prove lower/upper bounds, we argue that in fact succinct indexes are more appropriate to the design of a library of succinct tools for multiple usages than succinct integrated encodings, and that they are even directly required in certain applications. Some of the advantages of succinct indexes over succinct integrated encodings are:

(1) A succinct integrated 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 the required ADT. Thus when using succinct indexes, the given data can be either stored to achieve maximal compression or to achieve optimal support of the operations defined in the ADT.

(2) The existence of two succinct integrated encodings supporting different 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 a single succinct index that supports the union of the two corresponding sets of operations in a straightforward manner.

(3) In some cases, we need not store the data explicitly because it can be derived from some other information in a manner that efficiently supports 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[1]. Given the support for the interface of these ADTs, we can support an extended set of operations efficiently. The succinct indexes occupy small amounts of space compared to the information-theoretic lower bound for representing the given data. Our indexes for strings, binary relations and multi-labeled trees are presented in Sections 2.2, 3.2 and 4.2, respectively, and the background knowledge required to introduce these indexes are in Sections 2.1, 3.1 and 4.1.

Based on the succinct index for strings, we design (in Section 2.3.1) a succinct integrated encoding that represents a string of length n over an alphabet of size σ using

[1]We use $\log_2 x$ to denote the logarithmic base 2 and $\lg x$ to denote $\lceil \log_2 x \rceil$. Occasionally this matters.

$nH_k(S) + \lg\sigma \cdot o(n) + O(\frac{n\lg\sigma}{\lg\lg_\sigma})$ bits, which supports access/rank/select operations in $o((\lg\lg\sigma)^{1+\epsilon})$ time, for any fixed constant $\epsilon > 0$. Here H_k denotes the k^{th} order empirical entropy of a given string. We also design (in Section 2.3.2) a succinct text index using $nH_0(S) + O(\frac{n\lg\sigma}{\lg\lg_\sigma})$ bits that supports finding all the occ occurrences of a given pattern of length m in $O(m\lg\lg\sigma + \texttt{occ}\lg n/\lg^\epsilon\sigma)$ time, for any fixed constant $\epsilon > 0$. Our succinct integrated encodings of binary relations (in Section 3.3) and multi-labeled trees (in Section 4.3) occupy space close to the information-theoretic lower bounds.

2. STRINGS

We first design succinct indexes for a given string S of length n over alphabet $[\sigma]$. [2]

2.1. Background

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

2.1.1. Bit Vectors. A key structure for many succinct data structures, and for the research work in this paper, is a bit vector B of length n that supports ***rank*** and ***select*** operations. We assume that the positions in B are numbered $1, 2, \ldots, n$. For $\alpha \in \{0, 1\}$, we consider the following operations:

— $\texttt{bin\_rank}_B(\alpha, x)$, the number of occurrences of α in $B[1..x]$;
— $\texttt{bin\_select}_B(\alpha, r)$, 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 of succinctly representing bit vectors, in which part (a) is from Jacobson [1989] and Clark and Munro [1996], while part (b) is from Raman *et al.* [2007].

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.*

Although part (b) of the above lemma implies part (a), we cite both results here as using part (a) directly can simplify the space analysis in some proofs in this paper.

A less powerful version of $\texttt{bin\_rank}(1, x)$, denoted by $\texttt{bin\_rank}'(1, x)$, returns the number of 1s in $B[1..x]$ if $B[x] = 1$, and ∞ otherwise.

LEMMA 2.2 ([RAMAN ET AL. 2007]). *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.*

2.1.2. Strings. One can generalize the 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:

— $\texttt{string\_rank}_S(\alpha, x)$, the number of occurrences of character α in $S[1..x]$;
— $\texttt{string\_select}_S(\alpha, r)$, the position of the r^{th} occurrence of character α in the string;
— $\texttt{string\_access}_S(x)$, the character at position x in the string.

The subscript S is omitted when it is clear from the context.

Grossi *et al.* [2003] designed a data structure called ***wavelet tree*** that encodes a string using $nH_0 + o(n) \cdot \lg\sigma$ bits to support string_access, string_rank and string_select in $O(\lg\sigma)$ time. To design a succinct integrated encoding for strings over

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

large alphabets, Golynski *et al.* [2006] gave another encoding that uses $n \lg \sigma + O(\frac{n \lg \sigma}{\lg \lg \sigma})$ bits and supports `string_rank` and `string_access` in $O(\lg \lg \sigma)$ time, and `string_select` in constant time. The $\lg \lg \sigma$ factor in the above running time is more scalable for large alphabets than the $\lg \sigma$ factor of wavelet trees. However, their encoding is not compressed.

The above operations on strings have a number of applications [Grossi et al. 2003; Ferragina et al. 2004; He et al. 2005; Golynski et al. 2006; Barbay et al. 2007], and hence supporting them efficiently is a fundamental problem in the design of succinct data structures. Thus we design succinct indexes for strings to support these operations in Sections 2.2.

2.1.3. Permutations. One important data structure we use is a succinct representation of a permutation on $[n]$ that supports the efficient computations of the permutation and its inverse. It is fairly straightforward to represent a permutation π using $(1 + 1/s)n \lg n + O(n)$ bits to support $\pi()$ and $\pi^{-1}()$ in $O(s)$ time for any parameter $s > 0$. We simply store the forward permutation and an auxiliary structure that stores for every s^{th} position in every cycle of length greater than s, the element s positions earlier in that cycle. Munro *et al.* [2003] investigated this problem and trimmed the space required to $(1 + 1/s)n \log_2 n + O(n \lg \lg n / \lg n)$ bits.

To achieve this result, they explicitly encode the sequence $\pi(1), \pi(2), ..., \pi(n)$ in $n \log_2 n + o(n)$ bits, but only use the operator $\pi()$ to access the given data. Thus, this result can be rewritten in the form of designing succinct indexes:

LEMMA 2.3 ([MUNRO ET AL. 2003]). *Given support for $\pi()$ (or $\pi^{-1}()$) in $g(n)$ time on a permutation on $[n]$, there is a succinct index using $(n \lg n)/s + O(n \lg \lg n / \lg n)$ bits that supports $\pi^{-1}()$ (or $\pi()$) in $O(s \cdot g(n))$ time for any parameter $s > 0$.*

2.1.4. y-fast Tries. Another important data structure we use is a y-fast trie, proposed by Willard [1983] to encode a set E that consists of v distinct integers in the universe $[n]$ in $O(v \lg n)$ bits. It is an improvement in terms of space upon the ***stratified tree*** proposed by Van Emde Boas *et al.* [1977]. Given an integer x, the y-fast trie can be used to retrieve the largest integer in the set E that is less than or equal to x in $O(\lg \lg n)$ time. This operation is often called ***predecessor*** query.

If we treat the universe $[n]$ as a bit vector B of length n, and the v integers in the set as the positions of the 1s in B, the y-fast trie can encode B in $O(v \lg n)$ bits to support the retrieval of the position of the last 1 in $B[1..x]$ in $O(\lg \lg n)$ time. As the integers in the set E are stored in the leaves of a y-fast trie, if we store their ranks explicitly in the leaf nodes, we can augment the y-fast trie to support $\texttt{bin\_rank}_B(1, x)$ in $O(\lg \lg n)$ time using additional $v \lg n$ bits. More precisely, to compute $\texttt{bin\_rank}_B(1, x)$, we first locate the last 1 in $B[1..x]$ using the y-fast trie in $O(\lg \lg n)$ time, and then retrieve the rank stored in the corresponding leaf of the y-fast trie in constant time. Thus:

LEMMA 2.4 ([WILLARD 1983]). *A bit vector B of length n with v 1s can be encoded using $O(v \lg n)$ bits to support $\texttt{bin\_rank}_B(1, x)$ in $O(\lg \lg n)$ time.*

2.2. Succinct Indexes

In this section and the rest of this paper, we introduce succinct indexes in two steps: we first define the ADTs and then design succinct indexes for these ADTs.

In the case of strings, 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 at the cost of the space required to enumerate the characters used). 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 2.1.2 to include "negative" searches, we define a ***literal*** as either a character, $\alpha \in [\sigma]$, or its negation, $\bar{\alpha} \in [\sigma] - \{\alpha\}$ (we use the array notation for strings to refer to its characters and substrings):

Definition 2.5. Consider a string $S[1..n]$ over the alphabet $[\sigma]$. A position $x \in [n]$ ***matches literal*** $\alpha \in [\sigma]$ if $S[x] = \alpha$. A position $x \in [n]$ ***matches literal*** $\bar{\alpha}$ if $S[x] \neq \alpha$. For simplicity, we define $[\bar{\sigma}]$ to be the set $\{\overline{1}, \ldots, \overline{\sigma}\}$.

With this definition, we can use `string_rank` and `string_select` to perform negative searches. For example, given the string $bbaaacdd$, we have $\texttt{string\_rank}(\overline{a}, 7) = 4$, as there are 4 characters that are not a in the string up to position 7. We also have $\texttt{string\_select}(\overline{a}, 3) = 6$, as position 6 is the 3^{rd} position whose character is not a. It is clear that $\texttt{string\_rank}(\overline{\alpha}, x) = x - \texttt{string\_rank}(\alpha, x)$ for $\alpha \in [\sigma]$. In this paper, we only provide support for the $\texttt{string\_select}(\alpha, x)$ operator for the special case in which $x \in [\sigma]$, while providing efficient support for $\texttt{string\_select}(\alpha, x)$ when $x \in [\bar{\sigma}]$ is left as an open problem.

We also consider the following operations on strings in addition to the three primary operations introduced in Section 2.1.2:

Definition 2.6. 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 (and not including) 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 (and not including) position x, if it exists.

To illustrate the above two operations, consider the string $bbaaacdd$. We have that $\texttt{string\_pred}(a, 7) = 5$, 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. By allowing α to be possibly a literal in the set $[\bar{\sigma}]$, the α-predecessor/successor queries in fact generalize the colored predecessor/successor queries defined by Mortensen [2003; 2006].

We now design a succinct index to support rank/select operations on strings. We have the following result.

LEMMA 2.7. *Given support for* `string_access` *in $f(n, \sigma)$ time on a string $S \in [\sigma]^n$, there is a succinct index using $O(\frac{n \lg \sigma}{\lg \lg \lg \sigma})$ bits that supports* `string_rank` *for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ in $O((\lg \lg \lg \sigma)^2 (f(n, \sigma) + \lg \lg \sigma))$ 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 need only show how to support `string_rank` and `string_select` for $\alpha \in [\sigma]$.

First we conceptually treat the given string S and portions of S in several ways. We treat S as an $n \times \sigma$ table E with rows indexed by $1, 2, \ldots, \sigma$ and columns by $1, 2, \ldots, n$. 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. A ***chunk*** of S is a substring 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 $j \in [\sigma]$. Hence a chunk corresponds to a $\sigma \times \sigma$ segment of E, or σ equally spaced substrings of A. We denote the block corresponding to the α^{th} row of the segment of E corresponding to a chunk C by C_α, where $\alpha \in [\sigma]$. Figure 1 illustrates these concepts. In this example, let C be the 4^{th} chunk. Then we have $C_2 = 0011$.

We first construct a bit vector B which stores the cardinalities of all the blocks in unary (i.e. a block of cardinality l is stored as l 1s followed by a 0), in the order they

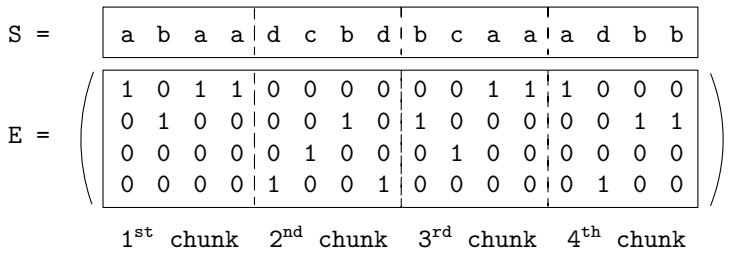

Fig. 1. A sample string for the proof of Lemma 2.7.

appear in A, so that $B = 1^{l_1}01^{l_2}0 \ldots 1^{l_n}0$, 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 the example in Figure 1, $B = 11100110101010101100101000110010$.

Using this 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 as suggested by Golynski *et al.* [2006]. To be specific, to compute $\texttt{string\_rank}_S(\alpha, x)$, let C be the chunk that position x is in (i.e. C is the u^{th} chunk, where $u = \lceil x/\sigma \rceil$). Since the number of occurrences of α in $S[1..x]$ is equal to the sum of the number of occurrences of α in the first $u-1$ chunks and the number of occurrences of α in $C[1..x \mod \sigma]$, we have $\texttt{string\_rank}_S(\alpha, x) = \texttt{string\_rank}_S(\alpha, (u-1)\sigma) + \texttt{string\_rank}_C(\alpha, x \mod \sigma)$. The first item on the right side of this equation can be computed using B as follows: Let a_1 and a_2 be the positions of the 0s in B that correspond to the last block in the $(\alpha-1)^{\text{st}}$ row of E, and the last block before block C_α in the α^{th} row of E, respectively. Then $a_1 = \texttt{bin\_select}_B(0, n(\alpha-1)/\sigma)$ and $a_2 = \texttt{bin\_select}_B(0, n(\alpha-1)/\sigma + u - 1)$. Since the number of occurrences of α in the first $u-1$ chunks is equal to the number of 1s before block C_α in the α^{th} row of E, the following equation holds: $\texttt{string\_rank}_S(\alpha, (u-1)\sigma) = \texttt{bin\_rank}_B(1, a_2) - \texttt{bin\_rank}_B(1, a_1)$. Therefore, we need only compute $\texttt{string\_rank}_C(\alpha, x \mod \sigma)$.

To compute $\texttt{string\_select}_S(\alpha, r)$, we first compute the position, v, of the 1 in B that corresponds to the r^{th} α in S. Let the number of 1s in $B[1..v]$ be q. As there are $\texttt{bin\_rank}_B(1, a_1)$ 1s in the top $(\alpha-1)$ rows of E, we have $q = \texttt{bin\_rank}_B(1, a_1) + r$. We can compute v in constant time using the equation $v = \texttt{bin\_select}_B(1, q)$. Let the block containing the q^{th} 1 in A be the y^{th} block in E in the row major order, and the chunk, C', containing the r^{th} α be the w^{th} chunk of the string S. Then $y = \texttt{bin\_rank}_B(0, v) + 1$ and $w = y - (\alpha-1)n/\sigma$. Thus we have $\texttt{string\_select}_S(\alpha, r) = \texttt{string\_select}_{C'}(\alpha, q - \texttt{bin\_rank}_B(1, \texttt{bin\_select}_B(0, y-1))) + (w-1)\sigma$. Hence we need only show how to support `string_rank` and `string_select` on a given chunk C.

We store the following data structures 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. $X = 1^{l_1}01^{l_2}0 \ldots 1^{l_\sigma}0$, where l_α is the number of 1s in the block C_α. There are σ 1s in X, each corresponding to a character of the chunk, and σ 0s, each corresponding to a block of the chunk. Hence the length of X is 2σ. We store it in $2\sigma + o(\sigma)$ bits using Part (a) of Lemma 2.1.
— We construct an array R such that $R[j] = \texttt{bin\_rank}_D(1, j) \mod k$, where D is the block $C_{C[j]}$, and k is a parameter which we fix later. Each element of R is an integer in the range $[0..k-1]$, so R can be stored in $\sigma \lg k$ bits.
— We construct a conceptual permutation π on $[\sigma]$, defined later in the proof. We store an auxiliary structure P that takes $O(\sigma \lg \sigma/s + \sigma \lg\lg \sigma / \lg \sigma)$ bits using Lemma 2.3,

where s is a parameter which we fix later, and supports access to π in $O(s \cdot g(n,\sigma))$ time, given $O(g(n,\sigma))$-time access to π^{-1}.

— For each block C_α in a chunk C, let F_α be a "sparsified" bit vector for C_α, in which only every k^{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 k). We encode F_α using Lemma 2.4 in $O(\lg\sigma \times l_\alpha/k)$ bits to support $\texttt{bin\_rank}_{F_\alpha}(1,i)$ in $O(\lg\lg\sigma)$ time. All the F_α's in a given chunk thus occupy $O((\sigma\lg\sigma)/k)$ bits in total.

We first show how to support $\texttt{bin\_rank}'(1,j)$ on block $D = C_{C[j]}$ (note that $D[j] = 1$; hence $\texttt{bin\_rank}'(1,j)$ is equivalent to $\texttt{bin\_rank}(1,j)$). For this, we first compute $C[j]$ using $\texttt{string\_access}$ in $f(n,\sigma)$ time. Then we compute $\texttt{bin\_rank}(1,j)$ on $F_{C[j]}$ in $O(\lg\lg\sigma)$ time, which is equal to $\lfloor \texttt{bin\_rank}'_D(1,j)/k \rfloor$. We also retrieve $R[j]$ in constant time, which is equal to $\texttt{bin\_rank}'_D(1,j) \mod k$. Since $\texttt{bin\_rank}'_D(1,j) = k\lfloor \texttt{bin\_rank}'_D(1,j)/k \rfloor + \texttt{bin\_rank}'_D(1,j) \mod k$, we can compute $\texttt{bin\_rank}'_D(1,j)$ in $O(f(n,\sigma) + \lg\lg\sigma)$ time.

The permutation π for a chunk C is 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 $\alpha = 1, 2, \cdots, \sigma$. For example, in Figure 1, let C be the 4^{th} chunk. Then $\pi = 1, 3, 4, 2$. Using π^{-1} to denote the inverse of π (in the previous example, $\pi^{-1} = 1, 4, 2, 3$), 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 computed using X in constant time, as it is equal to $\texttt{bin\_rank}_X(\texttt{bin\_select}_X(0, \alpha - 1))$, 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 $O((f(n,\sigma) + \lg\lg\sigma)s)$ time (note that the $f(n,\sigma) + \lg\lg\sigma$ term here comes from the time required to retrieve a given element of π^{-1}).

Golynski *et al.* [2006] showed how to support the $\texttt{string\_select}$ operation on a chunk C by a single access to π plus a few constant-time operations. This is achievable because π stores the positions of the occurrences of characters that appear in C. More precisely, to compute $\texttt{string\_select}_C(\alpha, r)$, we first compute the number, l_α, of occurrences of α in C. This can be computed by performing rank/select operations on X, since X encodes all the l_α's in unary, i.e. $l_\alpha = \texttt{bin\_rank}_X(1, \texttt{bin\_select}_X(0,\alpha)) - \texttt{bin\_rank}_X(1, \texttt{bin\_select}_X(0, \alpha - 1))$. We return ∞ if $l_\alpha < r$. Otherwise, we locate the 1 bit in X corresponding to the r^{th} occurrence of α in C, and then retrieve the result from C. More precisely, we have that $\texttt{string\_select}_C(\alpha, r) = \pi(\texttt{bin\_rank}_X(1, \texttt{bin\_select}_X(0, \alpha - 1)) + r)$. When combined with our approach, we can support $\texttt{string\_select}$ for any character $\alpha \in [\sigma]$ in $O((f(n,\sigma) + \lg\lg\sigma)s)$ time.

Golynski *et al.* [2006] also showed how to compute $\texttt{string\_rank}$ by calling $\texttt{string\_select}$ $O(\lg k)$ times. To be specific, to compute $\texttt{string\_rank}_C(\alpha, x)$, let $r_1 = k\lfloor \texttt{string\_rank}_C(\alpha, x)/k \rfloor$ and $r_2 = r_1 + k - 1$. We can compute r_1 and r_2 in $O(\lg\lg\sigma)$ time, as $\lfloor \texttt{string\_rank}_C(\alpha, x)/k \rfloor$ is equal to $\texttt{bin\_rank}(1, x)$ on F_α (i.e. the "sparsified" bit vector for the block C_α). As $r_1 \le \texttt{string\_rank}_C(\alpha, x) \le r_2$, we then perform a binary search in the range $[r_1..r_2]$. In each phase of the loop, we use $\texttt{string\_select}$ to check whether we have found the answer. Thus we can support operator $\texttt{string\_rank}$ in $O(s \lg k(f(n,\sigma) + \lg\lg\sigma))$ time.

As there are n/σ chunks, the sum of the space costs of the auxiliary structures constructed for all the chunks is clearly $O(n\lg k + n\lg\sigma(1/s + 1/k))$ bits. Choosing $s = \lg\lg\lg\sigma$ and $k = \lg\lg\sigma$ makes the overall space cost of all the auxiliary structures $O(\frac{n\lg\sigma}{\lg\lg\lg\sigma})$ bits. The query times for $\texttt{string\_rank}$ and $\texttt{string\_select}$ become $O((\lg\lg\lg\sigma)^2(f(n,\sigma) + \lg\lg\sigma))$ and $O(\lg\lg\lg\sigma(f(n,\sigma) + \lg\lg\sigma))$, respectively. □

We now extend our succinct indexes to support α-predecessor and α-successor queries.

LEMMA 2.8. *Using $n + o(n)$ additional bits, the succinct index of Lemma 2.7 also supports* `string_pred` *and* `string_succ` *for any character $\alpha \in [\sigma]$ in $O((\lg\lg\lg\sigma)^2(f(n,\sigma) + \lg\lg\sigma))$ time, and these two operators for any literal $\alpha \in [\bar{\sigma}]$ in $O(f(n,\sigma))$ time.*

PROOF. We only show how to support `string_pred`; `string_succ` can be supported similarly. For any character $\alpha \in [\sigma]$, it is clear that $\texttt{string\_pred}(\alpha, x) = \texttt{string\_select}(\alpha, \texttt{string\_rank}(\alpha, x) - 1)$. Thus the operators `string_pred` and `string_succ` can be supported for any character $\alpha \in [\sigma]$ in $O((\lg\lg\lg\sigma)^2(f(n,\sigma)+\lg\lg\sigma))$ time. Hence we need only show how to support $\texttt{string\_pred}(\alpha, x)$ when $\alpha \in [\bar{\sigma}]$.

For this we require another auxiliary structure. We construct a bit vector, U, of length n, in which $U[i] = 1$ iff $S[i] \neq S[i-1]$ (we set $U[1] = 1$). We store U using Part (a) of Lemma 2.1 in $n + o(n)$ 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 $S[x-1]$ using `string_access` in $f(n,\sigma)$ time. If $S[x-1] \neq c$, then we return $x-1$. Otherwise, let $j = \texttt{bin\_select}_U(1, \texttt{bin\_rank}_U(1, x-1))$. By the definition of U, we have $S[j-1] \neq S[j] = S[j+1] = \cdots = S[x-1] = c$. Therefore, $j-1$ is the answer, and it can be computed in $O(f(n,\sigma))$ time. □

Combining Lemmas 2.7 and 2.8, we have our first main result:

THEOREM 2.9. *Given support for* `string_access` *in $f(n,\sigma)$ time on a string $S \in [\sigma]^n$, there is a succinct index using $O(\frac{n\lg\sigma}{\lg\lg\lg\sigma})$ bits that supports:*

- `string_rank` *for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ in $O((\lg\lg\lg\sigma)^2(f(n,\sigma) + \lg\lg\sigma))$ time;*
- `string_select` *for any character $\alpha \in [\sigma]$ in $O(\lg\lg\lg\sigma(f(n,\sigma) + \lg\lg\sigma))$ time;*
- `string_pred` *and* `string_succ` *for any character $\alpha \in [\sigma]$ in $O((\lg\lg\lg\sigma)^2(f(n,\sigma) + \lg\lg\sigma))$ time, and these two operations for $\alpha \in [\bar{\sigma}]$ in $O(f(n,\sigma))$ 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 2.3.2. With this definition, we have:

THEOREM 2.10. *Given support for* `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 $O(\frac{n\lg\sigma}{\lg\lg\sigma})$ bits that supports* `string_rank`*,* `string_pred` *and* `string_succ` *for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$, as well as* `string_access`*, in $O(\lg\lg\sigma f(n,\sigma))$ time.*

PROOF. As in the proof of Lemma 2.7, we divide string S and its corresponding conceptual table E into chunks and blocks, and construct bit vector B for the entire string, and bit vector X and the auxiliary structure P for each chunk. We also store the set of "sparsified" bit vectors, F_α's, for each chunk. With the $f(n,\sigma)$-time support for `string_select` on S, using the method described in the proof of Lemma 2.7, we can support `string_rank` on S in $O(\lg\lg\sigma + \lg k f(n,\sigma))$ time.

Now we provide support for `string_access`. We first design the data structures supporting the access to π and π^{-1} for any chunk C (see the proof of Lemma 2.7 for the definition of π and π^{-1}). We assume that C is the i^{th} chunk of S. From the definition of π, we observe that if the r^{th} occurrence of α in C corresponds to the j^{th} 1 in X, then $\pi(j)$ is equal to $\texttt{bin\_select}(1, r)$ on the block C_α. Since X essentially stores the number of occurrence of each character in C in unary, we can compute α and r in $O(1)$ time by performing rank/select operations on X. More precisely, we can use the identities $\alpha = \texttt{bin\_rank}_X(0, \texttt{bin\_select}_X(1, j))$ + 1, and $r = \texttt{bin\_select}_X(1, j) - \texttt{bin\_select}_X(0, \alpha - 1)$.

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 $(i-1)\sigma$, we need only show how to compute z. Since B stores the cardinalities of the blocks in E, z can be computed by performing rank/select operations on B as follows: Let a_1 and a_2 be the positions of the 0s in B that correspond to the last block in the $(\alpha-1)^{\text{th}}$ row of E, and the block in the α^{th} row of E that ends at position $(i-1)\sigma$, respectively. Then $a_1 = \texttt{bin\_select}_B(0, (\alpha-1)n/\sigma)$ and $a_2 = \texttt{bin\_select}_B(0, (\alpha-1)n/\sigma + i - 1)$. As $z = \texttt{bin\_rank}_B(1, a_2) - \texttt{bin\_rank}_B(1, a_1)$, we can compute z in constant time. Thus we can compute $\pi(j)$ in $f(n, \sigma)$ time. With the auxiliary structure P, we can further compute any element of π^{-1} in $O(sf(n, \sigma))$ time by Lemma 2.3.

With the support for the access to $\pi^{-1}()$, we can now use the method of Golynski *et al.* [2006] to compute $C[j]$ as follows. We first compute $\pi^{-1}(j)$ in $O(sf(n, \sigma))$ time. By the definition of π^{-1}, we observe that the $(\pi^{-1}(j))^{\text{th}}$ 1 in X corresponds to $C[j]$. Thus $C[j] = \texttt{bin\_rank}_X(0, \texttt{bin\_select}_X(1, \pi^{-1}(j))) + 1$. Therefore, we can compute $C[j]$ in $O(sf(n, \sigma))$ time. Finally, by the identity $C[j] = S[(i-1)\sigma + j]$ for the i^{th} chunk of S, `string_access` can be supported in $O(sf(n, \sigma))$ time.

To analyze the space cost of the auxiliary data structures used to support the operations `string_rank` and `string_access` (B, X, y-fast tries, and P), we observe that the same data structures are defined in the proof of Lemma 2.9. Thus they occupy $O(n \lg \sigma(1/s + 1/k))$ bits. Choosing $s = \lg\lg\sigma$ and $k = \lg\sigma$ makes the overall space cost of all the auxiliary structures $O(\frac{n \lg \sigma}{\lg\lg\sigma})$ bits. The query time for `string_rank` and `string_access` becomes $O(\lg\lg\sigma f(n, \sigma))$.

The approach in the proof of Lemma 2.8 can be directly used to support the operations `string_pred` and `string_succ` in $O(\lg\lg\sigma f(n, \sigma))$ time with $n + o(n)$ additional bits. □

2.3. Applications

2.3.1. 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.* [2006] considered the problem and suggested a method with space requirements proportional to the k^{th} order entropy of a different but related string. Here we solve the problem in its original form.

In our solution, we make use of the result of Sadakane and Grossi [2006], who investigated the problem of encoding a string in its compressed form, while at the same time allowing efficient access to the string. Their main result is described in the following lemma:

LEMMA 2.11 ([SADAKANE AND GROSSI 2006]). *A string $S \in [\sigma]^n$ can be encoded using $nH_k(S) + O(\frac{n}{\log_\sigma n}(k \lg \sigma + \lg\lg n))$ bits*[3]*. When $k = o(\log_\sigma n)$, the above space cost is $nH_k(S) + \lg\sigma \cdot o(n)$ bits. This encoding can be used to retrieve any $O(\lg n)$ consecutive bits of the binary encoding of the string in $O(1)$ time.*

The problem addressed by the above lemma can be considered as a restricted version of the problem considered in this section. We now combine Lemma 2.11 with our succinct index for strings to solve the more general problem for which we provide support for rank/select operations in addition to random access to the string.

[3]González and Navarro [2006] noted that the term $(k \lg \sigma + \lg\lg n)$ appears erroneously as $(k + \lg\log_\sigma n)$ in [Sadakane and Grossi 2006].

THEOREM 2.12. *A string S of length n over alphabet $[\sigma]$ can be represented using $nH_k(S) + \lg\sigma \cdot o(n) + O(\frac{n\lg\sigma}{\lg\lg\lg\sigma})$ bits for any $k = o(\log_\sigma n)$, to support:*

— `string_access` *in $O(1)$ time;*
— `string_rank` *for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ in $O(\lg\lg\sigma(\lg\lg\lg\sigma)^2)$ time;*
— `string_select` *for any character $\alpha \in [\sigma]$ $O(\lg\lg\sigma\lg\lg\lg\sigma)$ time;*
— `string_pred` *and* `string_succ` *for any literal $\alpha \in [\sigma]$ in $O(\lg\lg\sigma(\lg\lg\lg\sigma)^2)$ time, and these two operations for any literal $\alpha \in [\bar{\sigma}]$ in $O(1)$ time.*

When $\sigma = O(\lg n/\lg\lg n)$, S can be represented using $nH_k(S) + \lg\sigma \cdot o(n) + O(n)$ bits to support the above operations in $O(1)$ time.

PROOF. We use Lemma 2.11 to store S in $nH_k(S) + O(\frac{n}{\log_\sigma n}(k\lg\sigma + \lg\lg n))$ bits. When $k = o(\log_\sigma n)$, the above space cost is $H_k(S) + \lg\sigma \cdot o(n)$. This representation allows us to retrieve any $O(\lg n)$ consecutive bits of the string in $O(1)$ time. Thus we can use it to retrieve $S[i]$ in $O(1)$ time (i.e. `string_access` can be supported in $O(1)$ time).

We store a succinct index for S using Theorem 2.9, and the support for the operations `string_rank`, `string_select`, `string_pred` and `string_succ` for arbitrary σ immediately follows. The overall space in bits is $nH_k(S) + \lg\sigma \cdot o(n) + O(\frac{n\lg\sigma}{\lg\lg\lg\sigma})$.

When $\sigma = O(\lg n/\lg\lg n)$, instead of constructing the entire succinct index for S, we construct the following auxiliary structures. We conceptually divide the string into chunks and blocks, and construct the bit vector B as in the proof of Lemma 2.7. This reduces the support for `string_rank` and `string_select` on S, to the support for these two operations on any given chunk C (see the proof of Lemma 2.7).

Let $l = \lfloor \log_2 n/(2\log_2\sigma)\rfloor$. We construct a table L, in which for each character $\alpha \in [\sigma]$, each integer $i \in [l]$, and each possible string $D \in [\sigma]^l$, we store the results of queries $\texttt{string\_rank}_D(\alpha, i)$ and $\texttt{string\_select}_D(\alpha, i)$ (i.e. $L[D, \alpha, i]$ stores $\texttt{string\_rank}_D(\alpha, i)$ and $\texttt{string\_select}_D(\alpha, i)$). There are $\sigma^l \le \sigma^{\log_2 n/(2\log_2\sigma)} = \sigma^{\frac{1}{2}\log_\sigma n} = \sqrt{n}$ different strings of length l over alphabet $[\sigma]$. We can store the result of each query above in $\lg(l+1)$ bits. Thus the table L occupies $O(\sigma \times l \times \sqrt{n} \times \lg(l+1)) = O(\lg n/\lg\lg n \times \sqrt{n} \times l\lg l) = O(\sqrt{n}\lg^2 n) = o(n)$ bits. Using the table L, we can answer queries $\texttt{string\_rank}(\alpha, i)$ and $\texttt{string\_select}(\alpha, i)$ on any string $D \in [\sigma]^l$ in constant time by performing a table lookup on L (as $L[D, \alpha, i]$ stores the answers). We can also support `string_rank` and `string_select` on any string G whose length, h, is less than l. This can be done by first appending the string with the first character till its length is l (on a word RAM, this step can be performed using a left shift of the binary encoding of G in constant time), and then use the resulting string, F, as a parameter to perform table lookups. Finally, as $\texttt{string\_rank}_G(\alpha, i) = \texttt{string\_rank}_F(\alpha, i)$ for $i \le h$, and $\texttt{string\_select}_G(\alpha, i) = \texttt{string\_select}_F(\alpha, i)$ if $\texttt{string\_select}_F(\alpha, i) \le h$ ($\texttt{string\_select}_G(\alpha, i) = \infty$ otherwise), we can support `string_rank` and `string_select` on G in constant time.

To support `string_rank` and `string_select` on any chunk C, we observe that $l = \Omega(\lg n/\lg\lg n)$. Therefore, the length of a chunk is either shorter than l, or can be divided into a constant number of substrings of length l and a substring of length at most l. To handle the latter case (the first case is already supported in the above paragraph), when answering $\texttt{string\_rank}_C(\alpha, i)$, we use table L to compute the number of α's in the substrings that appear before position i, and to compute $\texttt{string\_rank}(\alpha, i \bmod l)$ on the substring that contains position i. The sum of these values is the result. To compute $\texttt{string\_select}_C(\alpha, i)$, we computer the number of occurrences of α in each substring from left to right, and compute the subset sum, till we locate the substring that contains the result. We then use table lookup to retrieve the result.

To support string_pred and string_succ for $\alpha \in [\bar{\sigma}]$, we construct the bit vector U using $n+o(n)$ bits as in the proof of Lemma 2.8, and use the same algorithm to support string_pred and string_succ in constant time. The support for these two operators for $\alpha \in [\sigma]$ is straightforward.

The auxiliary data structures B, L and U occupy $O(n)$ bits in total, so the overall space cost is $nH_k(S) + \lg\sigma \cdot o(n) + O(n)$ bits. □

2.3.2. High-Order Entropy-Compressed Text Indexes for Large Alphabets. As a result of the growth of the textual data in databases, the World Wide Web and applications such as bioinformatics, various indexing techniques have been developed to facilitate full text searching. More formally, given a text string T of length n and a pattern string P of length m, whose symbols are drawn from the same fixed alphabet $[\sigma]$, 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 is contained 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 define ***pattern searching*** to be the process of answering all the above three types of queries for a given pattern string.

A suffix array [Manber and Myers 1993; Gonnet et al. 1992] is a popular type of text index. The idea is to organize the offsets of all the suffixes of the text in a sorted list using the suffixes as sort keys, which takes exactly $n \lg n$ bits. With a suffix array, one can answer existential and cardinality queries in $O(m \lg n)$ time, and listing queries in $O(\mathtt{occ})$ extra time. Additional information about the lengths of the (longest) common prefixes of pairs of suffixes of the text can be stored to speed up pattern search. By precomputing and storing such information for $2n-1$ pairs of suffixes (see [Gusfield 1997] for a detailed description of such pairs), one can answer existential and cardinality queries in $O(m+\lg n)$ time, and listing queries in $O(\mathtt{occ})$ extra time. Unfortunately, straightforward representation of such prefix length data takes $(2n-1)\lg n$ bits. Perhaps as a consequence, suffix arrays are less popular for large text collections. Other traditional text indexes have similar problems.

Researchers thus designed various succinct text indexes [Ferragina et al. 2004; Ferragina and Manzini 2005; Grossi and Vitter 2005; He et al. 2005; Golynski et al. 2006]. In this section, we apply our succinct indexes for strings to improve previous results. In particular, we design a high-order entropy-compressed text index.

We first present the following lemma to encode strings in zeroth order entropy while supporting rank and select:

LEMMA 2.13. *A string S of length n over alphabet $[\sigma]$ can be represented using $nH_0(S) + O(\frac{n\lg\sigma}{\lg\lg\sigma})$ 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 2.9, 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 $\lg\binom{n}{n_\alpha} + o(n_\alpha) + O(\lg\lg n)$ bits, where n_α is the number of occurrences of α in S. To analyze the space cost of all the $E[\alpha]$'s, we make use of Stirling's formula: $n! = \sqrt{2\pi n}(\frac{n}{e})^n(1+O(\frac{1}{n}))$. With this, we have $\log_2 n! =$

$n \log_2 n - n \log_2 e + \frac{1}{2} \log_2 n + O(1)$. We thus have:

$$\begin{aligned}
\lg \binom{n}{n_\alpha} &< \log_2 \binom{n}{n_\alpha} + 1 \\
&= \log_2 n! - \log_2 n_\alpha! - \log_2 (n - n_\alpha)! + 1 \\
&= n \log_2 n - n \log_2 e - n_\alpha \log_2 n_\alpha + n_\alpha \log_2 e - (n - n_\alpha) \log_2 (n - n_\alpha) \\
&+ (n - n_\alpha) \log_2 e + \frac{1}{2} (\log_2 n - \log_2 n_\alpha - \log_2 (n - n_\alpha)) + O(1) \\
&= n \log_2 n - n_\alpha \log_2 n_\alpha - (n - n_\alpha) \log_2 (n - n_\alpha) + O(1) \\
&= (n - n_\alpha) \log_2 n + n_\alpha \log_2 n - n_\alpha \log_2 n_\alpha - (n - n_\alpha) \log_2 (n - n_\alpha) + O(1) \\
&= n_\alpha \log_2 \frac{n}{n_\alpha} + (n - n_\alpha) \log_2 \frac{n}{n - n_\alpha} + O(1) \qquad (1)
\end{aligned}$$

To analyze the second term of equation 1, we rewrite it into $n_\alpha \log_2((1 + \frac{n_\alpha}{n - n_\alpha})^{\frac{n - n_\alpha}{n_\alpha}})$. As $(1+x)^{1/x}$ approaches e from below when x goes to 0, this term is at most to $n_\alpha \log_2 e$. With equation 1, we have:

$$\lg \binom{n}{n_\alpha} < n_\alpha \log_2 \frac{en}{n_\alpha} + O(1) \qquad (2)$$

Note that equation 2 is true even for the special case when $n_\alpha = 0$, if we follow the interpretation that $0 \log_2 0 = 1$ (this is used to define zeroth order entropy) to compute $n_\alpha \log_2 \frac{en}{n_\alpha} = -n_\alpha \log_2 \frac{n_\alpha}{en}$. Therefore, the space cost of $E[\alpha]$ in bits is less than $n_\alpha \log_2 \frac{en}{n_\alpha} + o(n_\alpha) + O(\lg \lg n)$. Using this to sum the space cost of all the $E[\alpha]$'s for $\alpha \in [\sigma]$, the last two terms sum to $O(n) + O(\sigma \lg \lg n) = O(n \lg \lg \sigma)$ (as $\sigma \leq n$), while the first term on the right-hand side sums to $nH_0(S) + n \lg e$. Therefore, the total space cost is at most $n(H_0(S) + O(\lg \lg \sigma))$ bits.

With the table E stored as above, `string_select` can be supported in $O(1)$ time, as $\texttt{string\_select}(\alpha, i) = \texttt{bin\_select}_{E[\alpha]}(1, i)$, for $\alpha \in [\sigma]$. With the constant-time support for `string_select` on S, we can construct a succinct index using Theorem 2.10 to support `string_rank` and `string_access` in $O(\lg \lg n)$ time. This index uses $n \cdot \lg \sigma / \lg \lg \sigma$ bits according to the proof of Theorem 2.10, so the overall space cost is $n(H_0(S) + O(\lg \sigma / \lg \lg \sigma))$ bits. □

Compared with Theorem 2.12, Lemma 2.13 compresses the string to zeroth order entropy instead of high-order entropy, but it supports navigational operations more efficiently. We will use this lemma to prove Theorem 2.15 which is our theorem on succinct text indexes.

Another tool we use to design our succinct text index is ***compression boosting*** [Giancarlo and Sciortino 2003; Ferragina et al. 2005]. The concept of compression boosting was proposed to design BWT-based compression algorithms to achieve good guaranteed compression performance. Given a compression algorithm that can store a string using space proportional to its zeroth order entropy, a compression booster can use it to compress the string in space proportional to its k^{th} order entropy. This is based on the claim that compressing a string, T, up to its k^{th} order entropy can be achieved by optimally partitioning its Burrows-Wheeler transformed string, $T^{\texttt{BWT}}$, and using a zeroth-order compressor to compress each partition [Giancarlo and Sciortino 2003]. One variant of the compression boosting technique that we use was proposed by Ferragina *et al.*:

LEMMA 2.14 ([FERRAGINA ET AL. 2004]). *Consider a compression algorithm A that can store any string S of length p in less than $pH_0(S) + f(p)$ bits, where $f(p)$ is*

a non-decreasing concave function. Given a text string T of length n drawn from alphabet $[\sigma]$, there is a partition, $S_1, S_2, \cdots, S_z$, of $T^{\mathtt{BWT}}$, such that, for any $k \leq 0$, we have

$$\sum_{i=1}^{z}(A(S_i)) \leq nH_k(T) + \sigma^k f(n/\sigma^k),$$

where $A(S_i)$ is the space (in bits) required to store S_i using algorithm A.
This partition can be computed in $O(n)$ time.

With the above results, we can now prove our theorem.

THEOREM 2.15. *A text string T of length n over alphabet $[\sigma]$ can be stored using $nH_k(T) + O(\frac{n \lg \sigma}{\lg \lg \sigma})$ bits for any $k \leq \beta \log_\sigma n - 1$ 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 $O(\lg n / \lg^\epsilon \sigma)$ time for any ϵ where $0 < \epsilon < 1$, and output a substring of length l in $O(l \lg \lg \sigma + \lg n / \lg^\epsilon \sigma)$ time.*

PROOF. As shown in [Ferragina et al. 2004; He et al. 2005], we can represent suffix arrays by encoding the Burrows-Wheeler transformed string, $T^{\mathtt{BWT}}$, of the raw text appropriately (see Sections 4.5, 4.6 and 4.7.2 in [He 2007] for a detailed description of these techniques). Ferragina *et al.* [2004] also presented how to design a high-order entropy-compressed text index given an encoding of $T^{\mathtt{BWT}}$ that occupies $nH_0(T^{\mathtt{BWT}})$ bits plus an appropriate lower order term. Here we borrow these ideas, and combine them with our results to prove this theorem.

We use Lemma 2.14 to partition $T^{\mathtt{BWT}}$ into a set of strings $S_1, S_2, \cdots, S_z$. We use Lemma 2.13 to encode each string. We construct a bit vector B of length n, in which $B[i] = 1$ iff there exists a string S_j, whose starting position is position i of $T^{\mathtt{BWT}}$. We encode B using part (b) of Lemma 2.1. We construct a two-dimensional array $M[1..z][1..\sigma]$, where $M[i][\alpha]$ stores the total number of occurrences of character α in strings $S_1, S_2, \cdots, S_{i-1}$. In addition, we construct an array N of size σ such that for each alphabet symbol α, $N[\alpha]$ stores the number of characters in the text that lexicographically precede it.

With the above data structures, we can compute $\mathtt{string\_rank}_{T^{\mathtt{BWT}}}(\alpha, i)$ as follows. Since the bit vector B records the starting positions of the strings S_j's, we can first locate the string S_j that position i is in using the identity $j = \mathtt{bin\_rank}_B(1, i)$. We also have position i of $T^{\mathtt{BWT}}$ is the q^{th} position of S_j, where $q = i - \mathtt{bin\_select}_B(1, j) + 1$. By the definition of $M[\alpha][j]$, we have $\mathtt{string\_rank}_{T^{\mathtt{BWT}}}(\alpha, i) = M[\alpha][j] + \mathtt{string\_rank}_{S_j}(\alpha, q)$. As $\mathtt{string\_rank}_{S_j}$ is supported in $O(\lg \lg \sigma)$ time, we can compute $\mathtt{string\_rank}_{T^{\mathtt{BWT}}}(\alpha, i)$ in $O(\lg \lg \sigma)$ time. We can also compute $\mathtt{string\_access}_{T^{\mathtt{BWT}}}(i)$ in $O(\lg \lg \sigma)$ time, as $T^{\mathtt{BWT}}[i] = \mathtt{string\_access}_{S_j}(q)$.

With the above operations supported, we now show how to answer queries. The main idea is to use the algorithms in [He et al. 2005; He 2007] directly (the parameters of auxiliary structures are slightly changed to achieve the desired results), and the more efficient support for $\mathtt{string\_rank}$ and $\mathtt{string\_access}$ operations on $T^{\mathtt{BWT}}$ allows us to provide faster support for pattern matching. More precisely, to answer existential and cardinality queries, we directly apply the ***backward search*** algorithm as in [He et al. 2005; He 2007]. At the i^{th} phase of the algorithm, we compute the range of the suffix array, *SA*, of T, in which all the corresponding suffixes are prefixed with $P[i..m]$. Each phase consists of $\mathtt{string\_rank}$ and $\mathtt{string\_access}$ operations on $T^{\mathtt{BWT}}$ plus some operations that take constant time. Therefore, existential and cardinality queries can be answered in $O(m \lg \lg \sigma)$ time. To answer listing queries, note that the backward search algorithm returns the range of *SA* in which all the corresponding suffixes are

prefixed with P. Thus it suffices to support the following operation: given a position in *SA*, compute the offset of the suffix of T that corresponds to it. For this we mark every $(\lg n/(\lg^\epsilon \sigma \lg\lg\sigma))^{\text{th}}$ position of T and store the offsets of these positions in an array I, sorted by lexicographic order of the suffixes starting at these positions. This occupies $\lg n \times n/(\lg n/(\lg^\epsilon \sigma \lg\lg\sigma)) = n\lg^\epsilon \sigma \lg\lg\sigma$ bits. We use an additional bit vector F of length n to indicate whether a given entry in *SA* points to a position that is stored in S, and store F using Part (b) of Lemma 2.1 in $o(n)$ bits (as there are $n\lg^\epsilon \sigma \lg\lg\sigma/\lg n < n\lg\lg n/\lg^{1-\epsilon} n$ 1s in F). With these data structures and the support of the operations stated in the previous paragraph, we can use the algorithm of He *et al.* [2005; 2007] to list each occurrence of P in $O(\lg n/\lg^\epsilon \sigma)$ time. Finally, to output a substring of length l starting at a given position, we store for every $(\lg n/(\lg^\epsilon \sigma \lg\lg\sigma))^{\text{th}}$ position in T, the index of its corresponding entry in *SA*. This auxiliary data structure of $n\lg^\epsilon \sigma \lg\lg\sigma$ bits allows us to directly apply the techniques of He *et al.* [2005; 2007] to output a substring of length l in $O(l\lg\lg\sigma + \lg n/\lg^\epsilon \sigma)$ time.

To analyze the space cost of all the data structures designed, the encodings of S_j's occupy $\sum_{i=1}^{z}(|S_i|H_0(S_i) + O(\lg|S_i|\lg\sigma/\lg\lg\sigma))$ bits in total. The encoding of B occupies $\lg\binom{n}{z} + o(n)$ bits. Array M occupies $z\sigma\lg n$ bits. Array N occupies $\sigma\lg n$ bits. All the other data structures occupy $O(n\lg^\epsilon \sigma\lg\lg\sigma)$ bits in total. Therefore, the total space cost in bits is:

$$
\begin{aligned}
&\sum_{i=1}^{z}(|S_i|H_0(S_i) + O(\lg|S_i|\lg\sigma/\lg\lg\sigma)) + z\sigma\lg n + \lg\binom{n}{z} + o(n)\\
&\quad + \sigma\lg n + O(n\lg^\epsilon \sigma\lg\lg\sigma)\\
&< \sum_{i=1}^{z}(|S_i|H_0(S_i) + O(\lg|S_i|\lg\sigma/\lg\lg\sigma)) + z(\sigma+1)\lg n + \sigma\lg n\\
&\quad + O(n\lg^\epsilon \sigma\lg\lg\sigma)\\
&= \sum_{i=1}^{z}(|S_i|H_0(S_i) + (\sigma+1)\lg n) + \sum_{i=1}^{z} O(\lg|S_i|\lg\sigma/\lg\lg\sigma) + \sigma\lg n\\
&\quad + O(n\lg^\epsilon \sigma\lg\lg\sigma)\\
&= \sum_{i=1}^{z}(|S_i|H_0(S_i) + (\sigma+1)\lg n) + O(n\lg\sigma/\lg\lg\sigma) + \sigma\lg n + O(n\lg^\epsilon \sigma\lg\lg\sigma).
\end{aligned}
$$

We then apply Lemma 2.14 to bound the above value by:

$$
\begin{aligned}
&nH_k(T) + O(\sigma^{k+1}\lg n) + O(n\lg\sigma/\lg\lg\sigma) + O(n\lg^\epsilon \sigma\lg\lg\sigma)\\
&= nH_k(T) + O(\sigma^{k+1}\lg n) + O(n\lg\sigma/\lg\lg\sigma).
\end{aligned}
\tag{3}
$$

When $k \le \beta\log_\sigma n - 1$ for $0 < \beta < 1$, we have $\sigma^{k+1} \le n^\beta$. In this case, the second item in equation 3 is bounded by $o(n)$, so equation 3 is bounded by $nH_k(T) + O(\frac{n\lg\sigma}{\lg\lg\sigma})$. □

Grossi *et al.* [2003] designed a text index that uses $nH_k + o(n)\cdot\lg\sigma$ bits, and supports existential and cardinality queries in $O(m\lg\sigma + \text{polylog}(n))$ time. Golynski *et al.* [2006] reduced the $\lg\sigma$ factor in the query time to a $\lg\lg\sigma$, but their index is not compressed. Our text index has the advantages of both these indexes.

3. BINARY RELATIONS

3.1. Background

Barbay *et al.* [2007] extended the problem of the encoding of strings to binary relations. Consider a binary relation R between a set of objects, $[n]$, and a set of labels, $[\sigma]$, under which each object can be associated with zero or more labels. Let t denote the number of object-label pairs. Thus R can be treated as t pairs from $[n]\times[\sigma]$. They considered the following operations:

— $\texttt{label\_rank}_R(\alpha, x)$, the number of objects labeled α up to (and including) object x;
— $\texttt{label\_select}_R(\alpha, r)$, the position of the r^{th} object labeled α;
— $\texttt{label\_access}_R(x, \alpha)$, whether object x is associated with label α;
— $\texttt{label\_nb}_R(\alpha)$, the number of objects associated with label α;
— $\texttt{object\_rank}_R(x, \alpha)$, the number of labels associated with object x up to (and including) label α;
— $\texttt{object\_select}_R(x, r)$, the r^{th} label associated with object x;
— $\texttt{object\_nb}_R(x)$, the number of labels associated with object x.

As before, the subscript R is omitted when it is clear from the context. Based on a reduction from the support of rank/select on binary relations to that on strings, Barbay *et al.* [2007] proposed an encoding of binary relations using $t(\lg\sigma + o(\lg\sigma))$ bits to support the operators `label_rank` and `label_access` in $O(\lg\lg\sigma)$ time, and `label_select` in constant time. Their encoding supports the operators `label_nb` and `object_nb` in $O(1)$ time, `object_rank` in $O((\lg\lg\sigma)^2)$ time, and `object_select` in $O(\lg\lg\sigma)$ time. They gave another encoding also using $t(\lg\sigma + o(\lg\sigma))$ bits that supports `label_nb`, `object_select` and `object_nb` in constant time, `label_select`, `object_rank` and `label_access` in $O(\lg\lg\sigma)$ time, and `label_rank` in $O(\lg\lg\sigma\lg\lg\lg\sigma)$ time.

3.2. Succinct Indexes

We consider a binary relation R, relating an object set $[n]$ and a label set $[\sigma]$, and containing t pairs. We assume that each object is associated with at least one label (thus $t \geq n$), and $n \geq \sigma$ (the converse is symmetric). We show how to extend the results to other cases by simple techniques after the proof of each Theorem. We define the interface of the ADT of a binary relation through the operator `object_select` defined in Section 2.1.2 that can be used to obtain the labels associated with a given object.

We generalize the definition of literals to binary relations:

Definition 3.1. Consider a binary relation formed by t pairs from an object set $[n]$ and a label set $[\sigma]$. An object $x \in [n]$ ***matches literal*** $\alpha \in [\sigma]$ if x is associated with α. An object $x \in [n]$ ***matches literal*** $\bar{\alpha} \in [\bar{\sigma}]$ if x is not associated with α.

We also generalize the definition of α-predecessor and α-successor to binary relations.

Definition 3.2. Consider a binary relation formed by t pairs from an object set $[n]$ and a label set $[\sigma]$, a literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ and an object $x \in [n]$. The ***α-predecessor of object*** x, denoted by $\texttt{label\_pred}(\alpha, x)$, is the last object matching α before (and not including) object x, if it exists. Similarly, the ***α-successor of object*** x, denoted by $\texttt{label\_succ}(\alpha, x)$, is the first object matching α after (and not including) object x, if it exists.

We have the following theorem:

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

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

Fig. 2. An example of the encoding of a binary relation.

THEOREM 3.3. *Given support for* `object_select` *in* $f(n,\sigma,t)$ *time on a binary relation* R *formed by* t *pairs from an object set* $[n]$ *and a label set* $[\sigma]$*, there is a succinct index using* $O(\frac{t\lg\sigma}{\lg\lg\lg\sigma})$ *bits that supports:*

— `label_rank` *for any literal* $\alpha \in [\sigma] \cup [\bar{\sigma}]$ *in* $O((\lg\lg\lg\sigma)^2(f(n,\sigma,t)+\lg\lg\sigma))$ *time;*
— `label_select` *for any label* $\alpha \in [\sigma]$ *in* $O((\lg\lg\lg\sigma)(f(n,\sigma,t)+\lg\lg\sigma))$ *time;*
— `label_pred` *and* `label_succ` *for any label* $\alpha \in [\sigma]$ *in* $O((\lg\lg\lg\sigma)^2(f(n,\sigma,t)+\lg\lg\sigma))$ *time, and these two operations for any literal* $\alpha \in [\bar{\sigma}]$ *in* $O(f(n,\sigma,t))$ *time;*
— `object_rank` *and* `label_access` *for any literal* $\alpha \in [\sigma] \cup [\bar{\sigma}]$ *in* $O((\lg\lg\lg\sigma)f(n,\sigma,t)+\lg\lg\sigma)$ *time;*
— `label_nb` *for any literal* $\alpha \in [\sigma] \cup [\bar{\sigma}]$ *and* `object_nb` *in* $O(1)$ *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 [Barbay et al. 2007] (See Figure 2 for an example):

— 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_select`. To compute the i^{th} character in `ROWS`, we need compute the corresponding object, x, and the rank, r, of the corresponding label among all the labels associated with x. The position of the 1 in `COLUMNS` corresponding to the i^{th} character in `ROWS` is $l = \texttt{bin\_select}_{\texttt{COLUMNS}}(1,i)$. Therefore, $x = \texttt{bin\_rank}_{\texttt{COLUMNS}}(0,l)+1$, and $r = l - \texttt{bin\_select}_{\texttt{COLUMNS}}(0,x-1)$ if $x > 1$ ($r = l$ otherwise). Thus with these additional operations, we can support `string_access` in $O(f(n,\sigma,t))$ time using one call to `object_select` in addition to some constant-time operations.

We store a succinct index for `ROWS` using Theorem 2.9 in $O(\frac{t\lg\sigma}{\lg\lg\lg\sigma})$ bits. As we can support `string_access` on `ROWS` using `object_access`, the index can support `string_rank` for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$, `string_pred` and `string_succ` for any character $\alpha \in [\sigma]$ in $O((\lg\lg\lg\sigma)^2(f(n,\sigma)+\lg\lg\sigma))$ time, `string_pred` and `string_succ` for any literal $\alpha \in [\bar{\sigma}]$ in $O(f(n,\sigma))$ time, and `string_select` for any character $\alpha \in [\sigma]$ in $O(\lg\lg\lg\sigma(f(n,\sigma)+\lg\lg\sigma))$ time. With this, we can use the approach of Barbay *et al.* [2007] to support `label_rank`, `label_select` and `label_access` operations on binary relations using rank/select on `ROWS` and `COLUMNS` as follows.

To compute $\texttt{label\_rank}(\alpha,x)$, we observe that the position of the 0 in `COLUMNS` that corresponds to the x^{th} column of E is $j = \texttt{bin\_select}_{\texttt{COLUMNS}}(0,x)$. Since each 1 stored in `COLUMNS` corresponds to a character of `ROWS`, the position of the last label associated with object x in `ROWS` is $k = \texttt{bin\_rank}_{\texttt{COLUMNS}}(1,j)$. As $\texttt{label\_rank}(\alpha,x) = \texttt{string\_rank}_{\texttt{ROWS}}(\alpha,k)$, we can support operation `label_rank` in $O((\lg\lg\lg\sigma)^2(f(n,\sigma,t)+\lg\lg\sigma))$ time.

To compute $\texttt{label\_select}(\alpha, r)$, we first observe that the position of the r^{th} occurrence of α in ROWS is $u = \texttt{string\_select}_{\texttt{ROWS}}(\alpha, r)$. The position of the 1 that corresponds to this character in COLUMNS is $v = \texttt{bin\_select}_{\texttt{COLUMNS}}(1, u)$, which corresponds to object $\texttt{bin\_rank}_{\texttt{COLUMNS}}(0, v) + 1$. This object is the answer. Thus label_select can be supported in $O(\lg\lg\lg\sigma(f(n, \sigma, t) + \lg\lg\sigma))$ time.

To compute $\texttt{object\_nb}(x)$, we observe that the result is the x^{th} number encoded in COLUMNS in unary. Thus $\texttt{object\_nb}(x) = \texttt{bin\_rank}_{\texttt{COLUMNS}}(1, \texttt{bin\_select}_{\texttt{COLUMNS}}(0, x)) - \texttt{bin\_rank}_{\texttt{COLUMNS}}(1, \texttt{bin\_select}_{\texttt{COLUMNS}}(0, x-1))$, so we can support object_nb in constant time. We can support label_nb for $\alpha \in [\sigma]$ in the same manner by encoding the number of objects associated with each label in unary in another bit vector W, which occupies $n + t + o(n + t)$ bits. For the example in Figure 2, $W = 1101011101110$. To support label_nb for $\alpha \in [\bar{\sigma}]$, we use the equation $\texttt{label\_nb}(\bar{c}) = n - \texttt{label\_nb}(c)$ for $c \in [\sigma]$.

To support object_rank, we construct, for each object y, a bit vector G_y of length σ, in which $G_y[\beta] = 1$ iff object y is associated with label β and $\texttt{object\_rank}(y, \beta)$ is divisible by $\lg\lg\sigma$. We encode G_y using Lemma 2.4. Let l_x be the number of labels associated with x. Then the number of 1s in G_y is $\lfloor l_x / \lg\lg\sigma \rfloor$. Hence G_y occupies $O(\lfloor l_x/\lg\lg\sigma\rfloor \times \lg\sigma) = O(l_x \lg\sigma/\lg\lg\sigma)$ bits, and the total space cost of all the G_y's is $O(t\lg\sigma/\lg\lg\sigma)$ bits. To compute $\texttt{object\_rank}(x, \alpha)$, let $r_1 = \lg\lg\sigma\lfloor \texttt{object\_rank}(x, \alpha)/\lg\lg\sigma\rfloor$ and $r_2 = r_1 + \lg\lg\sigma - 1$. We can compute r_1 and r_2 in $O(\lg\lg\sigma)$ time, as $\lfloor \texttt{object\_rank}(x, \alpha)/\lg\lg\sigma\rfloor$ is equal to $\texttt{bin\_rank}(1, \alpha)$ on G_x. As $r_1 \le \texttt{object\_rank}(x, \alpha) \le r_2$, we then perform a binary search in the range $[r_1..r_2]$. In each phase of the loop, we use object_select to check whether we have found the answer. Thus we can support operator object_rank in $O((\lg\lg\lg\sigma)f(n, \sigma, t) + \lg\lg\sigma)$ time.

To compute $\texttt{label\_access}(x, \alpha)$, we make use of the fact that object x is labeled α iff $\texttt{object\_rank}(x, \alpha) - \texttt{object\_rank}(x, \alpha - 1)$ is 1. Therefore, label_access can be supported in $O((\lg\lg\lg\sigma)f(n, \sigma, t) + \lg\lg\sigma)$ time.

It now suffices to design algorithms to support label_pred and label_succ. We show how to compute $\texttt{label\_pred}(\alpha, x)$; label_succ can be supported similarly. The position of the first label associated with x in ROWS is $p = \texttt{bin\_rank}_{\texttt{COLUMNS}}(1, \texttt{bin\_select}_{\texttt{COLUMNS}}(0, x-1)) + 1$. Thus the position of the last occurrence of character α in $\texttt{ROWS}[1..p-1]$ is $q = \texttt{string\_pred}_{\texttt{ROWS}}(\alpha, p)$, and the object associated with the label corresponding to this occurrence is $\texttt{bin\_rank}_{\texttt{COLUMNS}}(0, \texttt{bin\_select}_{\texttt{COLUMNS}}(1, q)) + 1$. This object is the answer. Hence we can support $\texttt{label\_pred}(\alpha, x)$ for $\alpha \in [\sigma]$ in $O((\lg\lg\lg\sigma)^2(f(n, \sigma, t) + \lg\lg\sigma))$ time, and the same operation for $\alpha \in [\bar{\sigma}]$ in $O(f(n, \sigma, t))$ time.

The space of the index is the sum of space cost of storing COLUMNS, W, G_y's and the index for ROWS, which is at most $n + t + o(n+t) + n + t + o(n+t) + t\lg\sigma/\lg\lg\sigma + O(\frac{t\lg\sigma}{\lg\lg\lg\sigma}) = O(\frac{t\lg\sigma}{\lg\lg\lg\sigma})$ bits. □

Note that the above approach also works without the assumption that each object is associated with at least one label, though we can not use the inequality $t \ge n$ to analyze the space cost. Thus without such an assumption, our succinct index occupies $O(\frac{t\lg\sigma}{\lg\lg\lg\sigma} + n)$ bits.

We compare our result with the information-theoretic lower bound of the representation of a binary relation. To compute the number of distinct binary relations formed by t pairs from an object set $[n]$ and a label set $[\sigma]$, we observe that the set of these t pairs is a subset of the set $[n] \times [\sigma]$. Hence there are $\binom{n\sigma}{t}$ such binary relations. Thus the information-theoretic lower bound of representing a binary relation is $\lg\binom{n\sigma}{t}$ bits. Barbay *et al.* [2007] showed that when the average number of labels associated with each object is small (more precisely, if $t/n = \sigma^{o(1)}$), the above lower bound is $t(\lg\sigma - o(\lg\sigma))$

bits. Therefore, the space cost of our succinct index is asymptotically smaller than the information-theoretic minimum space, when $t/n = \sigma^{o(1)}$ and $t \geq n$.

As we treat a binary relation as an $n \times \sigma$ boolean matrix with t 1s in the proof of Theorem 3.3, our result also applies to the problem of succinctly representing a boolean matrix to allow rank/select on rows and columns. By the analysis in the above paragraph, our solution is particularly space-efficient for sparse boolean matrices.

3.3. Applications

We now apply our succinct indexes to design a succinct integrated encoding for binary relations using space close to the information-theoretic lower bound.

THEOREM 3.4. *A binary relation R formed by t pairs from an object set $[n]$ and a label set $[\sigma]$ can be represented using $\lg \binom{n\sigma}{t} + O(\frac{t \lg \sigma}{\lg \lg \lg \sigma})$ bits to support:*

- `label_rank` *for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ in $O(\lg \lg \sigma (\lg \lg \lg \sigma)^2)$ time;*
- `label_select` *for any label $\alpha \in [\sigma]$ in $O(\lg \lg \sigma \lg \lg \lg \sigma)$ time;*
- `label_pred` *and* `label_succ` *for any label $\alpha \in [\sigma]$ in $O(\lg \lg \sigma (\lg \lg \lg \sigma)^2)$ time, and these two operations for any literal $\alpha \in [\bar{\sigma}]$ in $O(1)$ time;*
- `object_rank` *and* `label_access` *for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ in $O(\lg \lg \sigma)$ time;*
- `label_nb` *for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$,* `object_select` *and* `object_nb` *in $O(1)$ time.*

PROOF. We construct the bit vector `COLUMNS` as in the proof of Theorem 3.3 using $n + t + o(n + t)$ bits. We also construct another bit vector `BR`, which lists the bits of the conceptual table E (see the proof of Theorem 3.3 for the definition of E) in the column-major order. For the example in Figure 2, `BR` $= 00111001001011100001$. We store `BR` using Lemma 2.2 in $\lg \binom{n\sigma}{t} + o(t) + O(\lg \lg(n\sigma))$ bits.

To compute `object_select`(x, r), we need locate the row that contains the r^{th} 1 in the x^{th} column of E. The total number of 1s in columns $1, 2, ..., x-1$ of E is $i =$ `bin_rank`$_{\texttt{COLUMNS}}(1,$ `bin_select`$_{\texttt{COLUMNS}}(0, x-1))$. Thus, the r^{th} 1 in the x^{th} column of E is the $(r+i)^{\text{th}}$ 1 in `BR`, whose position in `BR` is $j =$ `bin_select`$_{\texttt{BR}}(1, r+i)$, which is in the $(j-(x-1)\sigma)^{\text{th}}$ row of E. Hence `object_select`$(x, r) = j - (x-1)\sigma$. Therefore, we can support `object_select` in constant time.

With the constant-time support for `object_select`, we can construct a succinct index for R using Theorem 3.3, and the support for the operations listed follows directly.

The overall space cost in bits is $n+t+o(n+t)+\lg \binom{n\sigma}{t}+o(t)+O(\lg \lg(n\sigma))+O(\frac{t \lg \sigma}{\lg \lg \lg \sigma}) = \lg \binom{n\sigma}{t} + O(\frac{t \lg \sigma}{\lg \lg \lg \sigma})$, as $t \geq n \geq \sigma$. □

As with Theorem 3.3, the above approach also works without the assumption that each object is associated with at least one label, though we can not use the inequality $t \geq n$ to analyze the space cost. Thus without such an assumption, our succinct representation occupies $\lg \binom{n\sigma}{t} + O(\frac{t \lg \sigma}{\lg \lg \lg \sigma} + n)$ bits. This is close to the information-theoretic minimum.

4. MULTI-LABELED TREES

4.1. Background

4.1.1. 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` (***depth first unary degree sequence***) order. The `DFUDS` sequence represents a node of degree d by d opening parentheses followed by a closing parenthesis. All the nodes are listed in preorder (an extra opening parenthesis is added to the beginning of the sequence), and

each node is numbered by its opening parenthesis in its parent's description (DFUDS number). See Figure 3 in Section 4 for an example.

Various succinct data structures were designed to represent ordinal trees [Jacobson 1989; Munro and Raman 2001; Geary et al. 2006; Benoit et al. 2005; Jansson et al. 2007]. Benoit *et al.* [2005] 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.* [2007] extended this representation to support a richer set of navigational operations. Some of the operations supported in [Benoit et al. 2005; Jansson et al. 2007] are (we refer to each node by its preorder number):

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

4.1.2. 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, i.e. each node is associated with an nonempty subset of $[n]$. 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. We assume that labeled/multi-labeled trees are ordinal trees.

Geary *et al.* [2006] defined labeled extensions of the first six operators defined in Section 4.1.1. Their data structures support those operators in constant time using simple auxiliary data structures to store label information in addition to their succinct ordinal tree representation [Geary et al. 2006]. However, the overall space required is $2n + n\big(\lg \sigma + O(\sigma \lg\lg\lg n / \lg\lg n)\big)$ bits, which is much more than the information-theoretic lower bound of $n \log_2 \sigma + 2n - O(\lg n)$ bits.

Ferragina *et al.* [2005] proposed another structure based on the xbw transform of a labeled tree, which conceptually builds a compressed suffix array for all the labeled rooted paths in the tree. It 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 about twice the minimum space required to encode the tree. Ferragina *et al.* [2005] also showed how to use a wavelet tree to reduce the size to $n \lg \sigma + O(n)$ bits, but each of the above operations then takes $O(\lg \sigma)$ time. This structure can be further compressed to $nH_k + O(n)$ bits, where H_k is the k^{th} order entropy of labeled trees they defined [Ferragina et al. 2005], based on the context of upward paths of the nodes. Another interesting operation supported by the above representation is the subpath query, which returns the number of nodes whose upward paths are prefixed with a given pattern. Given a pattern of length p, the above representation can answer the subpath query in $O(p \lg \sigma)$ time.

Based on the succinct integrated encoding for binary relations, Barbay *et al.* [2007] gave an encoding for labeled trees using $n\,(\lg \sigma + o(\lg \sigma))$ bits to support the retrieval of the descendants or (closest) ancestors by labels in $O(\lg\lg \sigma)$ time per node. It also supports the computation of the number of descendants (of a given node) associated with a given label in $O(\lg\lg \sigma)$ time. The same technique is generalized to represent multi-labeled trees in $t\,(\lg \sigma + o(\lg \sigma))$ bits and support the same operations.

4.1.3. Balanced Parentheses. One key structure used in many succinct tree representations is a balanced parenthesis sequence. Munro and Raman [2001] showed how to succinctly represent a balanced parenthesis sequence S of length $2n$ to support the following operations:

— $\texttt{rank\_open}_S(i)$ ($\texttt{rank\_close}_S(i)$), the number of opening (closing) parenthesis in the sequence up to (and including) position i;
— $\texttt{select\_open}_S(i)$ ($\texttt{select\_close}_S(i)$), the position of the i^{th} opening (closing) parenthesis in the sequence;
— $\texttt{find\_close}_S(i)$ ($\texttt{find\_open}_S(i)$), the matching closing (opening) parenthesis for the opening (closing) parenthesis at position i;
— $\texttt{excess}_S(i)$, the number of opening parentheses minus the number of closing parentheses in the sequence up to (and including) position i;
— $\texttt{enclose}_S(i)$, the closest enclosing (matching parenthesis) pair of a given matching parenthesis pair whose opening parenthesis is at position i.

The subscript S is omitted when it is clear from the context. Their result is:

LEMMA 4.1 ([MUNRO AND RAMAN 2001]). *A sequence of balanced parentheses S of length $2n$ can be represented using $2n + o(n)$ bits to support the operations* rank_open*,* rank_close*,* select_open*,* select_close*,* find_close*,* find_open*,* excess *and* enclose *in $O(1)$ time.*

4.2. Succinct Indexes

We now consider a multi-labeled tree. Recall that n denotes the number of nodes in the tree, $[\sigma]$ denotes the label alphabet, and t denotes the total number of node-label pairs. As with binary relations, we adopt the assumption that each node is associated with at least one label (thus $t \geq n$), and that $n \geq \sigma$. The results can be extended to other cases by simple reductions shown in Section 3 for binary relations. To design succinct indexes for 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 operations 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:

LEMMA 4.2. *Using the* DFUDS *representation [Benoit et al. 2005; Jansson et al. 2007], an ordinal tree with n nodes can be encoded in $2n + o(n)$ bits to support all the navigational operations defined in Section 4.1.1 and the following operations in $O(1)$ time:*

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

PROOF. As it is shown in [Benoit et al. 2005; Jansson et al. 2007] how to support all the navigational operations listed in the lemma except $\texttt{node\_rank}_{\texttt{DFUDS}}$ and $\texttt{node\_select}_{\texttt{DFUDS}}$, we need only provide support for these two operations. We use the operations supported by Lemma 4.1, as the DFUDS sequence is a balanced parenthesis sequence encoded using this lemma [Benoit et al. 2005; Jansson et al. 2007].

In the balanced parentheses representation of the DFUDS sequence of the tree [Benoit et al. 2005], each node corresponds to an opening parenthesis and a closing parenthesis, though not necessarily a matching pair. In the sequence, the opening parentheses

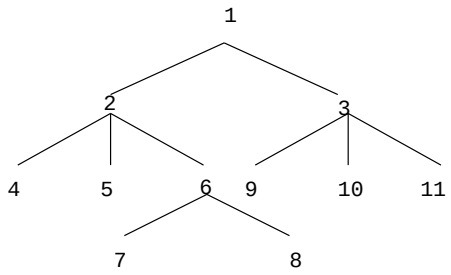

Fig. 3. An ordinal tree (where each node is assigned its rank in DFUDS order) and its DFUDS representation [Benoit et al. 2005].

correspond to nodes in DFUDS order, while the closing parentheses correspond to nodes in preorder. For example, in Figure 3, the 6^{th} node in DFUDS order (which is the 5th node in preorder) corresponds to the 6^{th} opening parenthesis, and the 5^{th} closing parenthesis.

With this observation, to compute `find_dfuds` for the node x (recall that it corresponds to the x^{th} closing parenthesis), we need to compute the rank of the corresponding opening parenthesis among all the opening parentheses. To compute this value, we consider the subsequence of the DFUDS representation of the tree that represents a node and all its descendants. In this subsequence, the number of closing parentheses minus the number of opening parentheses is equal to 1. Therefore, if x is the r^{th} child of its parent, then the closing parenthesis that comes before the DFUDS subsequence of node x matches the opening parenthesis that is r positions before the closing parenthesis in the DFUDS subsequence of x's parent. To make use of this fact, we first find the opening parenthesis that matches the closing parenthesis that comes before the DFUDS subsequence of node x. Its position in the sequence is $j = \texttt{find\_open}(\texttt{select\_close}(x-1))$. With j, we can compute the starting position of the subsequence of the parent of x, which is $p = \texttt{select\_close}(\texttt{rank\_close}(j)) + 1$, and $\texttt{child\_rank}(x)$, which is $r = \texttt{select\_close}(\texttt{rank\_close}(p)+1) - j$. Finally, $\texttt{rank\_open}(p+r-1)$ is the result.

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

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

Definition 4.3. 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 4.4. *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_select` *on R in $f(n, \sigma, t)$ time, there is a succinct index using $O(\frac{t \lg \sigma}{\lg \lg \lg \sigma})$ bits that supports on both R and $\pi(R)$:*

— `label_rank` *for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ in $O((\lg \lg \lg \sigma)^2 (f(n, \sigma, t) + \lg \lg \sigma))$ time;*
— `label_select` *for any label $\alpha \in [\sigma]$ in $O((\lg \lg \lg \sigma)(f(n, \sigma, t) + \lg \lg \sigma))$ time;*
— `label_pred` *and* `label_succ` *for any character $\alpha \in [\sigma]$ in $O((\lg \lg \lg \sigma)^2 (f(n, \sigma, t) + \lg \lg \sigma))$ time, and these two operations for any literal $\alpha \in [\bar{\sigma}]$ in $O(f(n, \sigma, t))$ time;*
— `object_rank` *and* `label_access` *for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ in $O((\lg \lg \lg \sigma) f(n, \sigma, t) + \lg \lg \sigma)$ time;*
— `label_nb` *for any literal $\alpha \in [\sigma] \cup [\bar{\sigma}]$ and* `object_nb` *in $O(1)$ time.*

PROOF. As $\texttt{object\_select}_{\pi(R)}(x, r) = \texttt{object\_select}_R(\pi^{-1}(x), r)$, we can support the operator `object_select` on $\pi(R)$ in $f(n, \sigma, t)$ time. Therefore, we can use Theorem 3.3 to construct succinct indexes for R and $\pi(R)$ and use the combined data structure to support the above operations. The total space cost is thus $O(\frac{t \lg \sigma}{\lg \lg \sigma})$ bits. □

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 [Zhang et al. 2006].

Definition 4.5. 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.* [2006] observed that in practice the document recursion level (when translated to our more precise definition, it is the maximum value of all ρ_αs 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 can now present our results:

THEOREM 4.6. *Consider a multi-labeled tree on n nodes and σ labels, associated in t pairs, of average recursivity ρ. Given support for* `node_label` *in $f(n, \sigma, t)$ time, there is a succinct index using $O(\frac{t \lg \sigma}{\lg \lg \sigma})$ bits that supports (for a given node x) the enumeration of:*

- *the set of α-descendants of x (denoted by D) in $O(|D|(\lg \lg \lg \sigma)^2(f(n, \sigma, t) + \lg \lg \sigma))$ time;*
- *the set of α-children of x (denoted by C) in $O(|C|(\lg \lg \lg \sigma)^2(f(n, \sigma, t) + \lg \lg \sigma))$ time;*
- *the set of α-ancestors of x (denoted by A) in $O((\lg \lg \lg \sigma)^2(f(n, \sigma, t) + \lg \lg \sigma) + |A|(\lg \lg \rho_\alpha + \lg \lg \lg \sigma(f(n, \sigma, t) + \lg \lg \sigma)))$ time using $t \lg \rho + O(\frac{t \lg \rho}{\lg \lg \rho})$ bits of extra space.*

PROOF. We encode the underlying ordinal tree structure in $2n + o(n)$ bits using Lemma 4.2. The sequence of nodes referred by their preorder (`DFUDS` order) numbers and the associated label sets form a binary relation R_p (R_d). Operations `find_dfuds` and `find_pre` provide constant-time conversions between the preorder numbers and the `DFUDS` order numbers, and `node_label` supports `object_access` on R_p. By Lemma 4.4, we can construct succinct indexes for R_p and R_d using $O(\frac{t \lg \sigma}{\lg \lg \sigma})$ bits, and support `label_rank`, `label_select` and `label_access` operations on either of them efficiently.

Using the technique of Barbay *et al.* [2007], the succinct index for R_p enables us to enumerate all the descendants of node x matching label α in $O(|D|(\lg \lg \lg \sigma)^2(f(n, \sigma, t) + \lg \lg \sigma))$ time (we can alternatively use the succinct index for R_d to achieve the same result). More precisely, we keep using `label_succ` to retrieve the nodes after but not including x that are associated with α, till we reach a node whose preorder number is greater than or equal to $(x + \texttt{desc\_nb}(x))$. Similarly, the succinct index of R_d enables us to enumerate all the children of node x matching α in $O(|C|(\lg \lg \lg \sigma)^2(f(n, \sigma, t) + \lg \lg \sigma))$ 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 (including 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 4.5 for the definitions of ρ_α and t_α), where the i^{th}

number of S_α corresponds to the i^{th} node labeled α in preorder. The lengths of the strings S_αs are implicitly encoded in R_p. We also encode for each label α its recursivity ρ_α in unary, using at most $t + \sigma + o(t + \sigma)$ bits, to provide constant-time access of each ρ_α. We use the encoding of Golynski *et al.* [2006](see Section 2.1.2) to encode each string S_α in $t_\alpha(\lg \rho_\alpha + O(\lg \rho_\alpha / \lg\lg \rho_\alpha))$ 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 $\sum_{\alpha\in[\sigma]} t_\alpha(\lg \rho_\alpha + O(\lg \rho_\alpha / \lg\lg \rho_\alpha))$ bits. By concavity of the logarithmic function and the function $g(n) = \lg n / \lg\lg n$, the above space cost is at most $\left(\sum_{\alpha\in[\sigma]} t_\alpha\right)\left(\lg\left(\frac{\sum_{\alpha\in[\sigma]} t_\alpha\rho_\alpha}{\sum_{\alpha\in[\sigma]} t_\alpha}\right) + O\left(f(\frac{\sum_{\alpha\in[\sigma]} t_\alpha\rho_\alpha}{\sum_{\alpha\in[\sigma]} t_\alpha})\right)\right) = t\lg\rho + O(\frac{t\lg\rho}{\lg\lg\rho})$.

To support the enumeration of all the α-ancestors of a node x, we first find from R_p the number, p_x, of nodes labeled α preceding x in preorder using `label_rank`. Then we iterate over i starting from 1. In each iteration, we first find the position p_i in S_α of the occurrence of i immediately preceding position p_x: it corresponds to the $p_i{}^{\text{th}}$ node labeled α in preorder (which 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 consists of a `label_select` operation 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 $O((\lg\lg\lg\sigma)^2(f(n,\sigma,t) + \lg\lg\sigma) + |A|(\lg\lg \rho_\alpha + (\lg\lg\lg\sigma)(f(n,\sigma,t) + \lg\lg\sigma)))$ 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 4.7. *The structure of Theorem 4.6 also supports, for any two given nodes x and y, the selection of:*

— *the first α-descendant of x after y in preorder in $O((\lg\lg\lg\sigma)^2(f(n,\sigma,t) + \lg\lg\sigma))$ time;*
— *the first α-child of x after y in preorder in $O((\lg\lg\lg\sigma)^2(f(n,\sigma,t) + \lg\lg\sigma))$ time;*
— *the first α-ancestor of x after y in preorder in $O((\lg\lg\lg\sigma)^2(f(n,\sigma,t) + \lg\lg\sigma) + \lg\lg \rho_\alpha)$ time.*

PROOF. Using the index in Theorem 4.6, we can easily support the first operation; it suffices to retrieve the first node labeled α after y using `label_succ` on R_p and then check whether it is a descendant of x. The support for the second operation is nontrivial only when y is a descendant of x (otherwise, the result is either the first α-child of x or ∞). In this case, we first locate the child, u, of x 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 `label_succ` on R_d.

To support the search for the first α-ancestor of x after y, we return ∞ if x precedes y in preorder. Otherwise, it suffices to consider the case when y is an ancestor of x, because 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 `label_succ` on the relation R_p and some navigational operators, we can find the first α-descendant z of y in preorder in $O((\lg\lg\lg\sigma)^2(f(n,\sigma,t) + \lg\lg\sigma))$ 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, which can be found in $O((\lg\lg\lg\sigma)^2(f(n,\sigma,t) + \lg\lg\sigma) + \lg\lg \rho_\alpha)$ time. □

The operations on multi-labeled trees are important for the support of XPath queries for XML trees [Barbay et al. 2007; Barbay 2006]. 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 encode different binary relations separately and thus waste a lot of space.

4.3. Applications

We now design a succinct integrated encoding of multi-labeled trees.

THEOREM 4.8. *Consider a multi-labeled tree on n nodes and σ labels, associated in t pairs, of average recursivity ρ. It can be represented using $\lg \binom{n\sigma}{t} + O(\frac{t \lg \sigma}{\lg \lg \sigma})$ bits to support (for a given node x) the enumeration of:*

- *the set of α-descendants of x (denoted by D) in $O(|D|(\lg \lg \lg \sigma)^2 \lg \lg \sigma)$ time;*
- *the set of α-children of x (denoted by C) in $O(|C|(\lg \lg \lg \sigma)^2 \lg \lg \sigma)$ time;*
- *the set of α-ancestors of x (denoted by A) in $O((\lg \lg \lg \sigma)^2 \lg \lg \sigma + |A|(\lg \lg \rho_\alpha + \lg \lg \sigma \lg \lg \lg \sigma))$ time using $t \lg \rho + O(\frac{t \lg \rho}{\lg \lg \rho})$ bits of extra space.*

It also supports (for any two given nodes x and y) the selection of:

- *the first α-descendant of x after y in preorder in $O((\lg \lg \lg \sigma)^2 \lg \lg \sigma)$ time;*
- *the first α-child of x after y in preorder in $O((\lg \lg \lg \sigma)^2 \lg \lg \sigma)$ time;*
- *the first α-ancestor of x after y in preorder in $O((\lg \lg \lg \sigma)^2 \lg \lg \sigma + \lg \lg \rho_\alpha)$ time using $t \lg \rho + O(\frac{t \lg \rho}{\lg \lg \rho})$ bits of extra space.*

PROOF. In the proofs of Theorem 4.6 and Corollary 4.7, we use Theorem 3.4 to encode the binary relation R_p, and construct a succinct index for R_d using Lemma 4.4. This theorem immediately follows. □

The discussions in Section 3.3 on the more general case of binary relations where each object may be associated with zero or more labels (instead of at least one label) also apply to the more general case for multi-labeled trees where each node may be associated with zero or more labels.

Using the approach in Section 2.3.1 , we can also 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. Although there is no standard definition for the entropy of binary relations or multi-labeled trees so that we cannot measure the compression theoretically, we can still achieve much compression in practice.

5. CONCLUSION

We have introduced the notion of succinct indexes for the design of data structures. We have shown their advantages by presenting succinct indexes for strings, binary relations, multi-labeled trees and multi-labeled graphs, and by applying them to various applications.

Using our techniques, we have designed a succinct encoding that represents a string of length n over an alphabet of size σ using $nH_k(S) + \lg \sigma \cdot o(n) + O(\frac{n \lg \sigma}{\lg \lg \lg \sigma})$ bits to support access/rank/select operations in $o((\lg \lg \sigma)^{1+\epsilon})$ time, for any fixed constant $\epsilon > 0$. This is the first compressed representation of strings supporting rank/select operations efficiently that occupies space proportional to the high-order entropies of strings. We have further designed a succinct text index using $nH_0(S) + O(\frac{n \lg \sigma}{\lg \lg \sigma})$ bits that supports pattern matching queries in $O(m \lg \lg \sigma + \mathtt{occ} \lg n / \lg^\epsilon \sigma)$ time, for a given pattern of length m. Previous results on this problem either have a $\lg \sigma$ factor instead of $\lg \lg \sigma$ in terms of running time [Grossi et al. 2003], or are not compressed [Golynski et al. 2006]. We have also designed a succinct encoding that represents a binary relation formed by

t pairs between n objects and σ labels using $\lg \binom{n\sigma}{t} + O(\frac{t \lg \sigma}{\lg \lg \lg \sigma})$ bits to support various types of rank/select operations efficiently. This space cost is close to the information-theoretic minimum. Our succinct representation of multi-labeled trees supports label-based ancestor, child and descendant queries at the same time, while previous results do not [Geary et al. 2006; Ferragina et al. 2005; Barbay et al. 2007].

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 one to design an encoding of the given data to achieve desired results or tradeoffs more easily, as the encoding need only support the ADT. In addition, to support new operations, it suffices 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. 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 separate 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.

There are some related open problems. First, it is not known whether the space costs of our succinct indexes are optimal. Thus one open problem is to prove tight lower bounds of the space required by succinct indexes of strings and binary relations, or to further improve the upper bounds. Second, the term $t \cdot o(\lg \sigma)$ of representing a binary relation in Theorem 3.4 is a second-order term only when $t/n = \sigma^{o(1)}$. Thus it is an open problem to reduce this term. Finally, for multi-labeled trees, as it requires $t \lg \rho + O(\frac{t \lg \rho}{\lg \lg \rho})$ bits of extra space to support the efficient retrieval of the α-ancestors of a given node, an open problem is to reduce this additional storage cost, or to prove that it is necessary.

REFERENCES

BARBAY, J. 2006. Adaptive search algorithm for patterns, in succinctly encoded XML. Tech. Rep. CS-2006-11, University of Waterloo, Ontario, Canada.

BARBAY, J., GOLYNSKI, A., MUNRO, J. I., AND RAO, S. S. 2007. Adaptive searching in succinctly encoded binary relations and tree-structured documents. *Theoretical Computer Science 387,* 3, 284–297.

BENOIT, D., DEMAINE, E. D., MUNRO, J. I., RAMAN, R., RAMAN, V., AND RAO, S. S. 2005. Representing trees of higher degree. *Algorithmica 43,* 4, 275–292.

CLARK, D. R. AND MUNRO, J. I. 1996. Efficient suffix trees on secondary storage. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*. 383–391.

DELPRATT, O., RAHMAN, N., AND RAMAN, R. 2006. Engineering the LOUDS succinct tree representation. In *Proceedings of the 5th International Workshop on Experimental Algorithms*. 134–145.

DEMAINE, E. D. AND LÓPEZ-ORTIZ, A. 2003. A linear lower bound on index size for text retrieval. *Journal of Algorithms 48,* 1, 2–15.

EMDE BOAS, P., KAAS, R., AND ZIJLSTRA, E. 1977. Design and implementation of an efficient priority queue. *Mathematical Systems Theory 10*, 99–127.

FERRAGINA, P., GIANCARLO, R., MANZINI, G., AND SCIORTINO, M. 2005. Boosting textual compression in optimal linear time. *Journal of the ACM 52,* 4, 688–713.

FERRAGINA, P., LUCCIO, F., MANZINI, G., AND MUTHUKRISHNAN, S. 2005. Structuring labeled trees for optimal succinctness, and beyond. In *Proceedings of the 46th IEEE Symposium on Foundations of Computer Science*. 184–196.

FERRAGINA, P. AND MANZINI, G. 2005. Indexing compressed text. *Journal of the ACM 52,* 4, 552–581.

FERRAGINA, P., MANZINI, G., MÄKINEN, V., AND NAVARRO, G. 2004. An alphabet-friendly FM-index. In *Proceedings of the 11th Symposium on String Processing and Information Retrieval*. Springer-Verlag LNCS 3246, 150–160.

GÁL, A. AND MILTERSEN, P. B. 2003. The cell probe complexity of succinct data structures. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*. 332–344.

GEARY, R. F., RAMAN, R., AND RAMAN, V. 2006. Succinct ordinal trees with level-ancestor queries. *ACM Transactions on Algorithms 2,* 4, 510–534.

GIANCARLO, R. AND SCIORTINO, M. 2003. Optimal partitions of strings: A new class of Burrows-Wheeler compression algorithms. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*. 129–143.

GOLYNSKI, A. 2007. Optimal lower bounds for rank and select indexes. *Theoretical Computer Science 387,* 3, 348–359.

GOLYNSKI, A., MUNRO, J. I., AND RAO, S. S. 2006. Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms*. 368–373.

GONNET, G. H., BAEZA-YATES, R. A., AND SNIDER, T. 1992. New indices for text: PAT trees and PAT arrays. In *Information Retrieval: Data Structures & Algorithms*, W. B. Frakes and R. A. Baeza-Yates, Eds. Prentice-Hall, Chapter 5, 66–82.

GONZÁLEZ, R. AND NAVARRO, G. 2006. Statistical encoding of succinct data structures. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching*. 294–305.

GROSSI, R., GUPTA, A., AND VITTER, J. S. 2003. High-order entropy-compressed text indexes. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*. 841–850.

GROSSI, R. AND VITTER, J. S. 2005. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing 35,* 2, 378–407.

GUSFIELD, D. 1997. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, New York, NY, USA.

HE, M. 2007. Succinct indexes. Ph.D. thesis, University of Waterloo.

HE, M., MUNRO, J. I., AND RAO, S. S. 2005. 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*. 23–32.

JACOBSON, G. 1989. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science*. 549–554.

JANSSON, J., SADAKANE, K., AND SUNG, W.-K. 2007. Ultra-succinct representation of ordered trees. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*.

MANBER, U. AND MYERS, E. W. 1993. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing 22,* 5, 935–948.

MILTERSEN, P. B. 2005. Lower bounds on the size of selection and rank indexes. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*. 11–12.

MORTENSEN, C. W. 2003. Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*. 618–627.

MORTENSEN, C. W. 2006. Fully dynamic orthogonal range reporting on RAM. *SIAM Journal on Computing 35,* 6, 1494–1525.

MUNRO, J. I., RAMAN, R., RAMAN, V., AND RAO, S. S. 2003. Succinct representations of permutations. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*. 345–356.

MUNRO, J. I. AND RAMAN, V. 2001. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing 31,* 3, 762–776.

RAMAN, R., RAMAN, V., AND SATTI, S. R. 2007. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms 3,* 4, 43.

SADAKANE, K. AND GROSSI, R. 2006. Squeezing succinct data structures into entropy bounds. In *Proceedings of the 17th annual ACM-SIAM Symposium on Discrete Algorithms*. 1230–1239.

WILLARD, D. E. 1983. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters 17,* 2, 81–84.

ZHANG, N., ÖZSU, M. T., ABOULNAGA, A., AND ILYAS, I. F. 2006. XSEED: Accurate and Fast Cardinality Estimation for XPath Queries. In *Proceedings of the 22nd International Conference on Data Engineering*. 61–72.

Received Month Year; revised Month Year; accepted Month Year