High-Order Entropy-Compressed Text Indexes

Roberto Grossi* Ankur Gupta† Jeffrey Scott Vitter‡

Abstract

We present a novel implementation of compressed suffix arrays exhibiting new tradeoffs between search time and space occupancy for a given text (or sequence) of n symbols over an alphabet Σ, where each symbol is encoded by $\lg|\Sigma|$ bits. We show that compressed suffix arrays use just $nH_h + O(n \lg\lg n / \lg_{|\Sigma|} n)$ bits, while retaining full text indexing functionalities, such as searching any pattern sequence of length m in $O(m \lg|\Sigma| + \text{polylog}(n))$ time. The term $H_h \leq \lg|\Sigma|$ denotes the hth-order empirical entropy of the text, which means that our index is nearly optimal in space apart from lower-order terms, achieving asymptotically the empirical entropy of the text (with a multiplicative constant 1). If the text is highly compressible so that $H_n = o(1)$ and the alphabet size is small, we obtain a text index with $o(m)$ search time that requires only $o(n)$ bits. Further results and tradeoffs are reported in the paper.

1 Introduction

The online proliferation of text documents and sequence data has made finding information difficult. Exhaustive search through very large collections is too slow. As a result, efficient indexing of massive documents is critical. Good indexes for textual and sequence data can be constructed quickly and offer a powerful tool set that allows fast access and queries on the text. However, storage cost can be significant because of the sheer volume of data; hence, we also want our text indexes to be a compact and space-efficient representation of the text.

We consider the text as a sequence T of n symbols, each drawn from the alphabet $\Sigma = \{0, 1, \ldots, \sigma\}$. The raw text T occupies $n \lg|\Sigma|$ bits of storage.[1] Without loss of generality, we can assume that $|\Sigma| \leq n$, since we only need to consider those symbols that actually occur in T. In practice, text is often compressible by a factor of 3 or 4, and sequence data may be more highly compressible. The text T is compressible if each symbol in the text can be represented, on average, in fewer than $\lg|\Sigma|$ bits. The empirical entropy H is the average number of bits per symbol needed to encode the text T. We trivially have $H \leq \lg|\Sigma|$.

For any integer $h \geq 0$, we can approximate the empirical entropy H of the text by the hth-order empirical entropy H_h, defined as

$$\begin{aligned} H_h &= \frac{1}{n} \sum_{x \in \Sigma^h} \lg \binom{n^x}{n^{x,1} n^{x,2} \ldots n^{x,|\Sigma|}} \\ &\sim \sum_{x \in \Sigma^h} \sum_{y \in \Sigma} -\frac{n^{x,y}}{n} \cdot \lg \frac{n^{x,y}}{n^x} \\ &= \sum_{x \in \Sigma^h} \sum_{y \in \Sigma} -\text{Prob}[y,x] \cdot \lg \text{Prob}[y|x] \leq \lg|\Sigma|, \end{aligned}$$

where n^x is the number of occurrences in the text of the h-symbol sequence x, for $0 \leq x \leq |\Sigma|^h - 1$, and $n^{x,y}$ represents the number of occurrences in the text of the concatenated sequence yx (y immediately preceding x), for $0 \leq y \leq |\Sigma| - 1$. Moreover, $\text{Prob}[y,x]$ represents the empirical joint probability that the symbol y occurs in the text immediately before the sequence x of h symbols, and $\text{Prob}[y|x]$ represents the empirical conditional probability that the symbol y occurs immediately before x, given that x occurs in the text.[2] High-order empirical entropy captures the dependence of symbols upon their context, and H_h converges to H for large h. In practice, once h gets larger than 4 or 5, the hth-order empirical entropy is close to the actual empirical entropy. Lempel and Ziv have provided an encoding such that $h \approx \alpha \lg n$ (where $0 < \alpha < 1$) is sufficiently good for approximating H with H_h; Luczak and Szpankowski prove a sufficient approximation when $h = O(\lg n)$ in [8].

In order to support fast searching, an index can be formed by preprocessing the text T. For any query

*Dipartimento di Informatica, Università di Pisa, via Filippo Buonarroti 2, 56127 Pisa (grossi@di.unipi.it). Support was provided in part by the University of Pisa.

†Center for Geometric and Biological Computing, Department of Computer Science, Duke University, Durham, NC 27708–0129 (agupta@cs.duke.edu). Support was provided in part by the ARO through MURI grant DAAH04–96–1–0013.

‡School of Science, Purdue University, West Lafayette, IN 47907–2068 (jsv@purdue.edu). Work was done in part while at Duke University. Support was provided in part by the Army Research Office through grant DAAD19–01–1–0725 and by the National Science Foundation through grant CCR–9877133.

[1] We use the notation $\lg_b^c n = (\lg n / \lg b)^c$ to denote the cth power of the base-b logarithm of n. Unless specified, we use $b = 2$.

[2] The standard definition considers the symbol y immediately *after* the sequence x, though it makes no difference.

pattern P of m symbols, the search task is to find P in T quickly. When the text is compressible, a natural question to ask is whether the index can be compressed by a similar amount and still support fast searching. Ideally, we would like to achieve nH bits, but all we can quantitatively analyze is nH_h, confining our algorithmic investigation to the hth-order empirical entropy. Since we want the text index to be as space-efficient as possible, our goal is nH_h bits and we want any amount over that to be as small as possible, preferably $o(n)$ bits. Note that we do not simply compress the text; we also support fast decompression and search of a portion of the text, *without* scanning the entire compressed text, which is unusual in classical compression schemes.

1.1 Related Work A new trend in the design of advanced indexes for full-text searching of documents is represented by compressed suffix arrays [6, 18, 19, 20] and opportunistic FM-indexes [2, 3], in that they support the functionalities of suffix arrays and suffix trees, which are more powerful than classical inverted files [4]. (An efficient combination of inverted file compression, block addressing and sequential search on word-based Huffman compressed text is described in [15].) They overcome the well-known space limitations by exploiting, in a novel way, the notion of text compressibility and the techniques developed for succinct data structures and bounded-universe dictionaries.

Grossi and Vitter [6] developed the compressed suffix array, a space-efficient incarnation of the standard suffix array. Using this structure, they implement compressed suffix trees in $2n \lg |\Sigma|$ bits in the worst case, with $o(m)$ searching time. If we index a 4-gigabyte ASCII file of Associated Press news in this manner, it requires 12 gigabytes, which includes explicit storage of the text.

The crucial notion of *self-indexing* text comes into play at this point. If the index is able to search for and retrieve any portion of the text *without* accessing the text itself, we no longer have to maintain the text in raw form—which can translate into a huge space savings. We refer to this type of index as a "self-index" to emphasize the fact that, as in standard text compression, it can replace the text. A self-index has the additional feature of avoiding a full scan of the compressed text for each search query.

Sadakane [19, 20] extended the functionalities in [6] to show that compressed suffix arrays are self-indexing, and he related the space bound to the order-0 empirical entropy H_0, getting the space bound $\epsilon^{-1} nH_0 + O(n \lg \lg |\Sigma|)$ bits, where $0 < \epsilon \leq 1$ and $\Sigma \leq \lg^{O(1)} n$. Searching takes $O(m \lg n)$ time. If we index the Associated Press file using Sadakane's index, we need roughly 1.6 gigabytes of storage, since we no longer have to store the text. However, the index is not as compressible as the text, even though it is still sublinear in the text size.

The FM-index [2, 3] is a self-indexing data structure using $5nH_h + O(n\frac{|\Sigma|+\lg\lg n}{\lg n} + n^\epsilon |\Sigma| 2^{|\Sigma| \lg |\Sigma|})$ bits, while supporting searching in $O(m + \lg^{1+\epsilon} n)$ time, where $|\Sigma| = O(1)$. It is based on the Burrows-Wheeler transform and is the first to encode the index size with respect to the high-order empirical entropy. In its space complexity, the second-order term may be $o(n)$ bits for small alphabet size $|\Sigma|$. Indexing the Associated Press file with the FM-index would require roughly 1 gigabyte according to the experiments in [3].

The above self-indexes are so powerful that the text is implicitly encoded in them and is not needed explicitly. Searching needs to decompress a negligible portion of the text and is competitive with previous solutions. In practical implementation, these new indexes occupy around 25–40% of the text size and do *not* need to keep the text itself. However, for moderately large alphabets, these schemes lose sublinear space complexity *even if* the text is compressible. Large alphabets are typical of phrase searching [5, 21], for example, in which the alphabet is made up of single words and its size cannot be considered a small constant.

1.2 Our Results In this paper, we develop self-indexing data structures that retain fast, full search functionality of suffix trees and suffix arrays, require roughly the same space as the optimally compressed version of the text to lower-order terms, and have less dependency on the alphabet size than previous methods [2, 3, 19, 20]. One of the contributions of this paper is that we shed light on the significance of the lower-order terms in the space complexity of self-indexes; these terms can dominate the empirical entropy cost when the alphabet grows moderately large. We also relate the issue of compressing a full-text index with high-order entropy to the succinct dictionary problem, in which t keys over a bounded universe n are stored in the information theoretically minimum space, $\lg \binom{n}{t}$ bits, plus lower-order terms (e.g., see [16, 17]).

Our main result is a new implementation of compressed suffix arrays that exhibits several tradeoffs between occupied space and search/decompression time. In one tradeoff, we can implement the compressed suffix array as a self-index requiring $nH_h + O(n \lg \lg n / \lg_{|\Sigma|} n)$ bits of space and allowing searches of patterns of length m in $O(m \lg |\Sigma| + \text{polylog}(n))$ time. Our method is the first self-index reaching asymptotically the high-order entropy, nH_h, of the text. The index is nearly optimal in space apart from lower-order terms, as the multiplicative constant in front of the high-order entropy term is 1. In some sense, we are in a similar situation to

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
T:	a	b	b	a	b	b	a	b	b	a	b	b	a	b	a	a	a	b	a	b	a	b	b	a	b	b	b	a	b	b	a	#
SA_0:	15	16	13	17	19	10	7	4	1	21	28	24	31	14	12	18	9	6	3	20	27	23	30	11	8	5	2	26	22	29	25	32
B_0:	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**1**	**1**	**0**	**0**	**1**
$rank(B_0, i)$:	0	1	1	1	1	2	2	3	3	3	4	5	5	6	7	8	8	9	9	10	10	10	11	11	12	12	13	14	15	15	15	16
Φ_0:	2	4	14	16	20	24	25	26	27	29	30	31	32	1	3	5	6	7	8	10	11	12	13	15	17	18	19	21	22	23	28	9

list a: $\langle 2, 4, 14, 16, 20, 24, 25, 26, 27, 29, 30, 31, 32 \rangle$, |list a| = 13
list b: $\langle 1, 3, 5, 6, 7, 8, 10, 11, 12, 13, 15, 17, 18, 19, 21, 22, 23, 28 \rangle$, |list b| = 18
list #: $\langle 9 \rangle$, |list #| = 1

Figure 1: Level $k = 0$ in the recursion for the compressed suffix array.

that of succinct dictionaries, achieving asymptotically the information theoretically minimum space.

In another tradeoff, we describe a different implementation of the compressed suffix array, occupying $\epsilon^{-1} nH_h + O(n \lg\lg n / \lg^{\epsilon}_{|\Sigma|} n)$ bits ($0 < \epsilon < 1/3$), so that searching a pattern of length m takes $O(m/\lg_{|\Sigma|} n + \lg^{\omega} n \lg^{1-\epsilon} |\Sigma|)$ time ($1 > \omega > 2\epsilon/(1-\epsilon) > 0$). By fixing $\epsilon = 1/4$, for example, we improve previous work both in terms of space and search time. When $nH_h = o(n)$ and alphabet size is small, we obtain the first self-index that *simultaneously* exhibits sublinear size $o(n)$ in bits and sublinear query time $o(m)$. More results are stated throughout the paper.

1.3 Preliminaries We use constant-time rank and select data structures [7, 13, 16]. For a bitvector B of size n, function $rank_{\mathbf{1}}(B, i)$ returns the number of **1**s in B up to (but not including) position i; the data structure requires $O(n \lg\lg n / \lg n)$ additional bits. The function $select_{\mathbf{1}}(B, i)$ returns the position of the ith **1** in B; the data structure requires $O(n/\lg\lg n)$ additional bits. We can also support $rank_{\mathbf{0}}$ and $select_{\mathbf{0}}$ using the same number of additional bits. Letting t be the number of elements stored (the number of **1**s in the bitvector), we can replace the bitvector supporting *rank* and *select* with the constant-time *indexable dictionaries* developed by Raman, Raman, and Rao [17], requiring $\lceil \lg \binom{n}{t} \rceil + O(t \lg\lg t / \lg t) + O(\lg\lg n)$ bits. If we also wish to support $rank_{\mathbf{0}}$ and $select_{\mathbf{0}}$, we can use the fully-indexable version of their structure, called an FID, requiring $\lceil \lg \binom{n}{t} \rceil + O(n \lg\lg n / \lg n)$ bits.

2 Compressed Suffix Arrays

A standard suffix array [4, 10] is an array containing the position of each of the n suffixes of text T in lexicographical order. In particular, $SA[i]$ is the starting position in T of the ith suffix in lexicographical order. The size of a suffix array is $\Theta(n \lg n)$ bits, as each of the positions stored uses $\lg n$ bits. A suffix array allows constant time *lookup* to $SA[i]$ for any i.

The compressed suffix array [6] contains the same information as a standard suffix array. The data structure is recursive in nature, where each of the $\ell = \lg\lg n$ levels indexes half the elements of the previous level. Hence, the kth level indexes $n_k = n/2^k$ elements:

1. Start with $SA_0 = SA$, the suffix array for text T.
2. For each $0 \le k < \lg\lg n$, transform SA_k into a more succinct representation through the use of a bitvector B_k, rank function $rank(B_k, i)$, neighbor function Φ_k, and SA_{k+1} (i.e, the recursion).
3. The final level, $k = \lg\lg n$ is written explicitly, using n bits.

SA_k is not actually stored (except at the last level ℓ), but we refer to it for the sake of explanation. B_k is a bitvector such that $B_k[i] = 1$ if and only if $SA_k[i]$ is even. Even-positioned suffixes are represented in SA_{k+1}, whereas odd-positioned suffixes are not. In order to retrieve odd-positioned suffixes, we employ the neighbor function Φ_k, which maps a position i in SA_k containing the value p into the position j in SA_k containing the value $p + 1$:

$$\Phi_k(i) = \{ \; j \quad \text{such that} \quad SA_k[j] = (SA_k[i] \bmod n) + 1 \; \}.$$

A *lookup* for $SA_k[i]$ can be answered as follows:

$$SA_k[i] = \begin{cases} 2 \cdot SA_{k+1}[rank_k(B_k, i)] & \text{if } B_k[i] = 1 \\ SA_k[\Phi_k(i)] - 1 & \text{if } B_k[i] = 0. \end{cases}$$

An example of the recursion for text T is given in Figure 1 for level $k = 0$, where a < b < # and # is a special end-of-text symbol. Here, $\Phi_0(4) = 16$, since $SA_0[4] = 17$ and $SA_0[16] = 17 + 1 = 18$. To retrieve $SA_0[16]$, since $B_0[16] = \mathbf{1}$, we compute $2 \cdot SA_1[rank(B_0, 16)] = 2 \cdot SA_1[12] = 2 \cdot 9 = 18$. To retrieve $SA_0[4]$, since $B_0[4] = \mathbf{0}$, we compute $SA_0[\Phi_0(4)] - 1 = SA_0[16] - 1 = 18 - 1 = 17$.

The representation of B_k and $rank(B_k, i)$ uses the methods of [7, 13, 16]. The major hurdle remains in the representation of Φ_k, which is at the heart of compressed suffix arrays. In fact, Φ_0 is the inverse of the LF mapping in the Burrows-Wheeler transform employed

in the FM-index [2, 3]. Grossi and Vitter developed an approach based on the notion of Σ *lists*. At level k, they partition the suffix pointers indexed ($n_k = n/2^k$ of them) by equal prefixes in the text of length 2^k. For $k = 0$ (level 0), the prefix length is $2^0 = 1$; thus they group the suffix pointers together according to the single symbol immediately preceding each suffix. For example, some position p is in list a if the suffix pointed to by $SA_0[p]$ in T is preceded by an a. For the text T in our example, the Σ lists for level $k = 0$ needed to represent Φ_k are shown in the bottom of Figure 1.

The fundamental property of these Σ lists is that each list is an increasing sequence of positions from the text. Compression is achieved since we can simply encode, for each list, the lengths of the gaps in the increasing sequence. By concatenating the lists together (called L_k in [6]), adding some header and directory information, and encoding the gaps of L_k, we can achieve constant time access to $\Phi_k(i)$ by looking at the ith entry in the concatenated list L_k (see [6]).

Sadakane [19, 20] has proposed an alternative method to represent the Σ lists that can be bounded in terms of the zero-order empirical entropy H_0. In order to achieve self-indexing, he also defines the notion of the *inverse* suffix array SA^{-1}, such that $SA^{-1}[j] = i$ if and only if $SA[i] = j$. To compress SA^{-1} along with SA, it suffices to keep SA_ℓ^{-1} in the last level ℓ. The cost of a *lookup* for SA^{-1} is the same as that for SA. Sadakane describes a *substring* algorithm that, given an index i of the compressed suffix array, decompresses c consecutive symbols from the text starting from position $SA[i]$ by incrementally applying Φ_0 each time, for a total cost of $O(c)$ time. It is not difficult to extend his method to Φ_k for any k, such that each application of Φ_k decompresses $\Theta(2^k)$ symbols, for a total cost of $O(c/2^k + \lg^\epsilon n)$ time. We use the inverse compressed suffix array and this extended version of decompression in Section 5. In future sections, we assume the following set of operations.

DEFINITION 2.1. ([6, 19, 20]) *Given a text T of length n, a* compressed suffix array *for T supports the following operations without requiring explicit storage of T or its (inverse) suffix array:*

- compress *produces a compressed representation that encodes (i) text T, (ii) its suffix array SA, and (iii) its inverse suffix array SA^{-1};*
- lookup *in SA returns the value of $SA[i]$, the position of the ith suffix in lexicographical order, for $1 \le i \le n$;* lookup *in SA^{-1} returns the value of $SA^{-1}[j]$, the rank of the jth suffix in T, for $1 \le j \le n$;*
- substring *decompresses the portion of T corresponding to the first c symbols (a prefix) of the suffix in $SA[i]$, for $1 \le i \le n$ and $1 \le c \le n - SA[i] + 1$.*

3 Higher-Order Entropy Representation of Compressed Suffix Array

In this section, we further reduce the space in the compressed suffix array to the size of the text in compressed form, replacing the $\lg|\Sigma|$ factor in the space bound to H_h, for any $h \le \alpha \lg_{|\Sigma|} n$ with $0 < \alpha < 1$. (Luczak and Szpankowski [8] show that the average phrase length of the Lempel-Ziv encoding is $O(\lg n)$ bits.) We get this reduction by restructuring in a novel way Φ_k, which contributes the bulk of the space that compressed suffix arrays use.

Our basic idea is to partition each Σ list y further into sublists $\langle x, y\rangle$, using an h-symbol prefix x (hereafter known as *context*) of each of the suffixes. (The $|\Sigma|^{2^k}$-symbol sequence y precedes the context x in the text.) For example, for context length $h = 2$ and level $k = 0$, if we continue the above example, we break the Σ lists by context (in lexicographical order aa, ab, a#, ba, bb, and # and numbered from 0 to $|\Sigma|^h - 1$) as shown below. We use # to represent $\#^j$ for all $j < h$. (Only level $k = 0$ is shown here.) Each sublist $\langle x, y\rangle$ we have created is also increasing. The lists are stored in the nonempty entries of the columns in the table below, partitioned by row according to contexts x of length 2:

x	list a	list b	list #
aa	$\langle 2\rangle$	$\langle 1\rangle$	$\emptyset$
ab	$\langle 4\rangle$	$\langle 3,5,6,7,8,10,11,12\rangle$	$\langle 9\rangle$
a#	$\emptyset$	$\langle 13\rangle$	$\emptyset$
ba	$\langle 14,16,20\rangle$	$\langle 15,17,18,19,21,22,23\rangle$	$\emptyset$
bb	$\langle 24,25,26,27,29,30,31\rangle$	$\langle 28\rangle$	$\emptyset$
#	$\langle 32\rangle$	$\emptyset$	$\emptyset$

The crucial observation to make is that all entries in the row corresponding to a given context x create a *contiguous* sequence of positions from the suffix array. For instance, along the fourth row of the table above for $x =$ ba, there are 10 entries that are contiguous and in the range [14, 23]. The conditional probability that a precedes context ba is 3/10, that b precedes context ba is 7/10, while that of # preceding context ba is 0. As a result, we show in Section 3.5 that encoding each of these sublists efficiently using context statistics results in a code length related to H_h, the hth-order entropy. In exchange, we must store more complicated context information. Below, we give some intuition about the information needed and how our method represents it in a compact way.

x	n_0^x	$\#x$	list a	list b	list #
aa	2	0	$\langle 2\rangle$	$\langle 1\rangle$	$\emptyset$
ab	10	2	$\langle 2\rangle$	$\langle 1,3,4,5,6,8,9,10\rangle$	$\langle 7\rangle$
a#	1	12	$\emptyset$	$\langle 1\rangle$	$\emptyset$
ba	10	13	$\langle 1,3,7\rangle$	$\langle 2,4,5,6,8,9,10\rangle$	$\emptyset$
bb	8	23	$\langle 1,2,3,4,6,7,8\rangle$	$\langle 5\rangle$	$\emptyset$
#	1	31	$\langle 1\rangle$	$\emptyset$	$\emptyset$

For level $k = 0$ above, n_0^x is the number of elements in each context (row) x, and $\#x$ represents the partial sum of all prior entries; that is, $\#x = \sum_{x'<x} n_0^{x'}$. Using these arrays, we transform the standard lists by *renumbering each element based on its order within its context.* For example, the final entry in list b, 28, is written as 5 in the new list b, as it is the fifth element in context bb. We can recreate b's final entry 28 from this information since we also store the number of elements in all prior contexts (in lexicographical order). For instance, we can recreate $\Phi_0(31) = 28$ by accessing 5 from the new list b and adding $\#x = 23$, the number of elements in all contexts prior to $x =$ bb.

3.1 Preliminary Notions As we saw before, n_k^x is the number of elements at level k that are in context x. Similarly, we can define n_k^y as the number of elements at level k in list y; and $n_k^{x,y}$ as the number of elements at level k that are in both context x and list y, that is, the size of sublist $\langle x, y\rangle$. For instance, in the example, $n_0^{\mathtt{b}} = 18$ and $n_0^{\mathtt{ab,b}} = 8$. (Note that $\sum_x n_k^x = n_k$, $\sum_y n_k^y = n_k$, and $\sum_{x,y} n_k^{x,y} = n_k$.) Notice that $H_h \sim \sum_{x\in\Sigma^h} \sum_{y\in\Sigma} -(n_0^{x,y}/n_0)\lg(n_0^{x,y}/n_0^x)$, where $n_0 = n$.

In what follows, we define s_k to be the number of nonempty sublists $\langle x, y\rangle$ at level k. We bound $s_k \le \min\{|\Sigma|^{2^k+h}, n_k\}$ as follows. The first inequality $s_k \le |\Sigma|^{2^k+h}$ is true since there are at most $|\Sigma|^h$ contexts x and at most $|\Sigma|^{2^k}$ lists y at level k. The second inequality $s_k \le n_k$ comes from the worst case where each sublist contains only a single element.

We define $\ell = \lg\lg n$ to be the last level in the compressed suffix array, as in Section 2. We introduce a special level $\ell' < \ell$, such that $2^{\ell'} = O(\lg_{|\Sigma|} n)$ and $2^{\ell-\ell'} = O(\lg|\Sigma|)$. Rather than storing all of the ℓ levels as discussed in Section 2, we only maintain the recursion explicitly until level ℓ', where $2^{\ell'} = O(\lg_{|\Sigma|} n)$ and $2^{\ell-\ell'} = O(\lg|\Sigma|)$. In this section, we prove:

LEMMA 3.1. *We can implement compressed suffix arrays using* $nH_h \lg\lg_{|\Sigma|} n + 2(\lg e+1)n + O(n\lg\lg n/\lg n)$ *bits and* $O(n\lg|\Sigma|)$ *preprocessing time for compress, so lookup takes* $O(\lg\lg_{|\Sigma|} n + \lg|\Sigma|)$ *time and substring for* c *symbols takes the cost of lookup plus* $O(c/\lg_{|\Sigma|} n)$ *time.*

LEMMA 3.2. *For any fixed value of* $0 < \epsilon < 1/2$, *we can implement compressed suffix arrays using* $\epsilon^{-1}nH_h + 2(\lg e+1)n + O(n\lg\lg n/\lg n)$ *bits and* $O(n\lg|\Sigma|)$ *preprocessing time for compress, so that lookup takes* $O\big((\lg_{|\Sigma|} n)^{\epsilon/(1-\epsilon)} + \lg|\Sigma|\big)$ *time and substring for* c *symbols takes the cost of lookup plus* $O(c/\lg_{|\Sigma|} n)$ *time.*

3.2 Overview of the Method for Representing Φ_k In order to support a query for $\Phi_k(i)$, we need the following basic information: the list y containing $\Phi_k(i)$, the context x containing $\Phi_k(i)$, the element z stored explicitly in list y, and the number of elements $\#x$ in all contexts prior to x. In the example from Section 2, for $\Phi_0(31) = 28$, we have $y =$ b, $x =$ bb, $\#x = 23$, and $z = 5$. The value for $\Phi_k(i)$ is then $\#x + z$, as shown in the example. We execute five main steps to answer a query for $\Phi_k(i)$ in constant time:

1. Consult a directory G_k to determine $\Phi_k(i)$'s list y and the number of elements in all prior lists, $\#y$. (We now know that $\Phi_k(i)$ is the $(i-\#y)$th element in list y.) In the example above, we consult G_0 to find $y =$ b and $\#y = 13$.
2. Consult a list L_k^y to determine the context x of the $(i-\#y)$th element in list y. For example, we consult $L_0^{\mathtt{b}}$ to determine $x =$ bb.
3. Look up the appropriate entry in list y to find z. In particular, we look in the sublist of list y having context x, which we call the $\langle x, y\rangle$ *sublist* and encode efficiently. In the example, we look for the appropriate element in the $\langle$bb, b$\rangle$ sublist and determine $z = 5$.
4. Consult a directory F_k to determine $\#x$, the number of elements in all prior contexts. In the example, after looking at F_0, we determine $\#x = 23$.
5. Return $\#x + z$ as the solution to $\Phi_k(i)$. The example code would then return $\Phi_k(i) = \#x + z = 23 + 5 = 28$.

We now detail each step given above.

3.3 Directories G_k and F_k We describe the details of the directory G_k (and the analogous structure F_k) defined in Section 3.2, which determines $\Phi_k(i)$'s list y and the number of elements in all prior lists $\#y$. We can think of G_k conceptually as a bitvector of length n_k, the number of items indexed at level k. For each nonempty list y at level k (considered in lexicographical order) containing n_k^y elements, we write $(n_k^y - 1)$ **0**s, followed by a **1**. Intuitively, each **1** represents the last element of a list. Since there are as many **1**s in G_k as nonempty lists at level k, G_k cannot have more than $\min\{|\Sigma|^{2^k}, n_k\}$ **1**s. To retrieve the desired information in constant time, we compute $y = rank(G_k, i)$ and $\#y = select(G_k, y)$. The F_k directory is similar, where each **1** denotes the end of a context (considered in lexicographical order), rather than the end of a list. Since there are at most $\min\{|\Sigma|^h, n_k\}$ possible contexts, we have at most that many **1**s. We use the indexable dictionary method [17] for this purpose.

LEMMA 3.3. *Let* $M(r)$ *denote* $\min\{|\Sigma|^r, n_k\}$. *We can store* G_k *using* $O(M(2^k)\lg\frac{n}{M(2^k)})$ *bits of space, and* F_k *using space* $O(M(h)\lg\frac{n}{M(h)})$.

3.4 List-Specific Directory L_k^y Once we know which list our query $\Phi_k(i)$ is in, we must find its context x. We create a directory L_k^y for each list y at level k, exploiting the fact that the entries are grouped into $\langle x, y\rangle$ sublists as follows. We can think of L_k^y conceptually as a bitvector of length n_k^y, the number of items indexed in list y at level k. For each nonempty $\langle x, y\rangle$ sublist (considered in lexicographical order) containing $n_k^{x,y}$ elements (where $\sum_x n_k^{x,y} = n_k^y$), we write $(n_k^{x,y} - 1)$ **0**s, followed by a **1**. Intuitively, each **1** represents the last element of a sublist. Since there are as many **1**s in L_k^y as nonempty sublists in list y, that directory cannot have more than $\min\{|\Sigma|^h, n_k^y\}$ **1**s. Directory L_k^y is made up of two distinct components:

The first component is an indexable dictionary [17] in the style of Section 3.3 that produces a nonempty context number $p \geq 0$. In the example given in Section 2, context **ba** has $p = 2$ in list **a**, and $p = 3$ in list **b**. It also produces the number $\#p$ of items in all prior sublists, which is needed in Section 3.5. In the example given above, context **ba** has $\#p = 2$ in list **a**, and $\#p = 10$ in list **b**. To retrieve the desired information in constant time, we compute $p = \mathit{rank}(L_k^y, i - \#y)$ and $\#p = \mathit{select}(L_k^y, p)$.

In order to save space, for each level k, we actually store a single directory shared by all lists y. For each list y, it can retrieve the list's p and $\#p$ values. Conceptually, we represent this global directory L_k as a simple concatenation of the list-specific bitvectors described above. The only additional information we need is the starting position of each of the above bitvectors, which is easily obtained by computing $\mathit{start} = \#y$. We compute $p = \mathit{rank}(i) - \mathit{rank}(\mathit{start})$ and $\#p = \mathit{select}(p + \mathit{rank}(\mathit{start}))$. L_k is implemented by a single indexable dictionary [17] storing s_k entries in a universe of size n_k.

LEMMA 3.4. *We can compute p and $\#p$ at level k in constant time using $O(s_k \lg \frac{n_k}{s_k})$ bits.*

The second component maps p, the local context number for list y, into the global one x (for level k). Since there are at most $\min\{|\Sigma|^h, n_k\}$ different contexts x for nonempty sublists $\langle x, y\rangle$ and at most $\min\{|\Sigma|^{2^k}, n_k\}$ nonempty lists y at level k, we use the concatenation of $|\Sigma|^{2^k}$ bitvectors of $|\Sigma|^h$ bits each, where bitvector b_k^y corresponds to list y and its **1**s correspond to the nonempty sublists of list y. We represent the concatenation of bitvectors b_k^y in lexicographical order by y using a single indexable dictionary. Mapping a value p to a context x for a particular list y is equivalent to identifying the position of the pth **1** in b_k^y. This can be done by a constant number of *rank* and *select* queries.

LEMMA 3.5. *We can map p to x at level k in constant time and $O(s_k \lg \frac{|\Sigma|^{2^k+h}}{s_k})$ bits.*

3.5 Encoding Sublist Gaps Armed with x, y, $\#y$, and $\#p$, we can now retrieve z, the encoded element we want from the $\langle x, y\rangle$ sublist. We compute z by finding the $(i - \#y - \#p)$th element in the $\langle x, y\rangle$ sublist. We now show how to get constant time access to any element in the sublist as well as relate the encoding size to H_h.

We use the indexable dictionary [17] to encode each $\langle x, y\rangle$ sublist, storing $n_k^{x,y}$ items (the number of items in the $\langle x, y\rangle$ sublist at level k) out of n_k^x (the number of items in context x at level k). The resulting bound is

$$\left\lceil \lg \binom{n_k^x}{n_k^{x,y}} \right\rceil + O\left(\frac{n_k^{x,y} \lg\lg n_k^{x,y}}{\lg n_k^{x,y}}\right) + O\left(\lg\lg n_k^x\right),$$

where $\left\lceil \lg \binom{n_k^x}{n_k^{x,y}} \right\rceil \sim n_k^{x,y} \lg\left(e \frac{n_k^x}{n_k^{x,y}}\right) = n_k^{x,y} \lg\left(\frac{n_k^x}{n_k^{x,y}}\right) + n_k^{x,y} \lg e$, which encodes precisely the hth-order entropy, representing the ratio of elements in context x to those in both context x and list y, plus an additive term of $n_k^{x,y} \lg e$. Said another way, we are encoding the probability of being in list y given the knowledge that we are in context x, which again, precisely represents the hth-order entropy term.

LEMMA 3.6. *We can encode all sublists at level k using at most $nH_h + n_k \lg e + O(n_k \lg\lg(n_k/s_k)/\lg(n_k/s_k) + O(s_k \lg\lg(n_k/s_k))$ bits per level.*

3.6 Resulting Bounds for the Compressed Suffix Array We have almost all the pieces we need to prove Lemma 3.1 and Lemma 3.2. We store all levels $k = 0, 1, \ldots, \ell' - 1, \ell', \ell$ of the recursion in the compressed suffix array (notice the gap between ℓ' and ℓ). For each of the levels up to ℓ', we store a bitvector B_k and a neighbor function Φ_k as described in Section 2, with their space detailed below.

- Bitvector B_k stores $n_k/2$ (or $n_\ell \leq n_k/2$ when $k = \ell'$) entries out of n_k. It is implemented with an indexable dictionary [17] in $n_k/2 + O(n_k \lg\lg n_k / \lg n_k)$ bits.
- Neighbor function Φ_k is implemented as described in Sections 3.2–3.5, with the space bounds stated in Lemmas 3.3–3.6.

Similar to what we described in Section 2, level $k = \ell$ stores the suffix array SA_ℓ, inverted suffix array SA_ℓ^{-1}, and an array LCP_ℓ storing the longest common prefix information [10] to allow fast searching in SA_ℓ. Each array contains $n/\lg n$ entries, which implies that they occupy $3n$ bits in total; however, this can be reduced to less than n bits with minor modifications. In particular, we can simply increase the last level from ℓ to $\ell + 2$

without changing the overall space complexity. In sum, we obtain the following upper bound on the total number of bits: of space required for the compressed suffix array: $nH_h \lg \lg_{|\Sigma|} n + 2(\lg e + 1)n + O(\frac{n \lg \lg n}{\lg n})$.

Building the above data structures is simply a variation of what was done in [6]; thus it takes $O(n \lg |\Sigma|)$ time to *compress* (as given in Definition 2.1). Since accessing any of the data structures in any level requires constant time, *lookup* requires $O(\ell' + 2^{\ell - \ell'}) = O(\lg \lg_{|\Sigma|} n + \lg |\Sigma|)$ time, and a *substring* query for c symbols requires $O(c + \lg \lg_{|\Sigma|} n + \lg |\Sigma|)$ time. However, we can improve *substring* by noting that $\Phi_{\ell'}$ decompresses $2^{\ell'} = \Theta(\lg_{|\Sigma|} n)$ symbols at a time, thus we can replace the c term above by $c/\lg_{|\Sigma|} n$ as we remarked in Section 2. This completes the proof of Lemma 3.1.

In order to prove Lemma 3.2, we just store a constant number $1/\epsilon$ of the ℓ' levels as in [6], where $0 < \epsilon \leq 1/2$. In particular, we store level 0, level ℓ', and then one level every other $\gamma\ell'$ levels; in sum, $1 + 1/\gamma = 1/\epsilon$ levels, where $\gamma = \epsilon/(1-\epsilon)$ with $0 < \gamma < 1$. Each such level $k \leq \ell'$ stores

- a directory D_k (analogous to B_k) storing the $n_{k+\gamma\ell'} < n_k/2$ (or $n_l < n_k/2$ when $k \leq \ell'$) entries of the next sampled level. It is similarly implemented with an indexable dictionary [17] in less than $n_k/2 + O(n_k \lg \lg n_k / \lg n_k)$ bits; and
- a neighbor function Φ_k implemented as before.

The last level ℓ stores the arrays SA_ℓ, SA_ℓ^{-1}, and LCP_ℓ as previously described in less than n bits. (Recall that we actually store them at level $\ell + 2$.) We can bound the total number of bits required by the compressed suffix array as $\epsilon^{-1} nH_h + 2(\lg e + 1)n + O(n \lg \lg n / \lg n)$. Thus, we are able to save space at a small cost to *lookup*, namely, $O(\epsilon^{-1} 2^{\gamma\ell'} + 2^{\ell - \ell'}) = O(\lg^{\gamma}_{|\Sigma|} n + \lg |\Sigma|)$ time, while *substring* for c symbols requires an additional $O(c/\lg_{|\Sigma|})$ time. Building the above data structures is again a variation of what was done in [6], so *compress* requires $O(n \lg |\Sigma|)$ time, proving Lemma 3.2.

4 Compressed Suffix Arrays in Sublinear Space

We can further reduce the space in the implementation of compressed suffix arrays described in Section 3 to get a sublinear space bound by applying an even more succinct encoding of the Σ lists.

THEOREM 4.1. *Compressed suffix arrays can be implemented using $\epsilon^{-1} nH_h + O(n \lg \lg n / \lg^{\epsilon}_{|\Sigma|} n)$ bits and $O(n \lg |\Sigma|)$ preprocessing time for compress, so lookup takes $O\big((\lg_{|\Sigma|} n)^{\epsilon/1-\epsilon} \lg |\Sigma|\big)$ time and substring for c symbols takes the cost of lookup plus $O(c/\lg_{|\Sigma|} n)$ time.*

By a simple modification, we obtain an interesting special case:

THEOREM 4.2. *Compressed suffix arrays can be implemented using $nH_h + O(n \lg \lg n / \lg_{|\Sigma|} n)$ bits and $O(n \lg |\Sigma|)$ preprocessing time for compress, so that lookup takes $O(\lg^2 n / \lg \lg n)$ time and substring for c symbols takes the cost of lookup plus $O(c \lg |\Sigma|)$ time.*

REMARK 4.1. *The compressed suffix array in Theorem 4.2 is a nearly space-optimal self-index in that it uses nearly the same space as the* compressed *text—nH_h bits—plus lower-order terms. For example, when Σ is of constant size, we get $nH_h + O(n \lg \lg n / \lg n)$ bits.*

4.1 Space Redundancy of Multiple Directories

In previous results, we encoded each sublist containing k items as a separate dictionary using roughly $\lg \binom{n}{k} \sim k \lg(en/k)$ bits. In total among all the sublists at level k, the total space bound was more than the entropy term nH_h by an additive term of $n_k \lg e$. However, in some sense, a dictionary (or sublist) is an optimal encoding of the locations of a particular symbol (or list) in the text T, so it should not require more space per symbol than the entropy representation, which is the optimal encoding size. This apparent paradox can be resolved by noting that each dictionary only represented the locations of one particular symbol in a sequence; that is, there was a separate dictionary for each symbol. In entropy encoding, we encode the locations of all the symbols together. Thus, it is more expensive to have a dictionary for the locations of *each symbol individually* rather than having a single encoding for the entire sequence of symbols.

Let's define t_k^x to be the number of nonempty sublists contained in context x at level k. We define $t_k = \max\{t_k^1, t_k^2, \ldots, t_k^{|\Sigma|^h}\}$ to be the maximum number of sublists that appear in a single context at level k. For a given context x at level k, let the number of entries in the nonempty sublists be $n_k^{x,1}, n_k^{x,2}, \ldots, n_k^{x,t_k^x}$. Given x, the entropy of encoding the preceding 2^k symbols y is

$$\frac{1}{n_k^x} \lg \binom{n_k^x}{n_k^{x,1}, n_k^{x,2}, \ldots, n_k^{x,t_k^x}} \sim \frac{1}{n_k^x} \sum_{1 \leq y \leq t_k^x} n_k^{x,y} \lg \frac{n_k^x}{n_k^{x,y}}. \tag{4.1}$$

On the other hand, if for context x we store a dictionary of the locations of the entries in each sublist, then the dictionary for a given sublist y in context x must encode

1. each of the $n_k^{x,y}$ occurrences of y, which happen with probability $n_k^{x,y}/n_k^x$.
2. each of the $n_k^x - n_k^{x,y}$ non-occurrences of y, which happen with probability $1 - n_k^{x,y}/n_k^x$.

The resulting entropy of encoding the preceding 2^k symbols y in context x is $\frac{1}{n_k^x} \sum_{1 \leq y \leq t_k^x} \lg \binom{n_k^x}{n_k^{x,y}} < \lg e + \frac{1}{n_k^x} \sum_{1 \leq y \leq t_k^x} n_k^{x,y} \lg \frac{n_k^x}{n_k^{x,y}}$. The $\lg e$ additive term in the number of bits per sublist occurrence is the extra cost

incurred by encoding all positions where the sublist does *not* occur. When summed over all n_k entries in all sublists and all contexts, this gives an $n_k \lg e$ contribution to the total space bound. Note that the $\lg e$ term does not appear in (4.1). We can remove this undesired space term at the cost of a more expensive lookup by encoding each of our sublists in terms of the locations not occupied by prior sublists within the same context. For example, at level $k = 0$, the $\langle x, \mathtt{a}\rangle$ sublist occupies various locations among the items in context x. When we encode the $\langle x, \mathtt{b}\rangle$ sublist, we only encode the positions that the $\langle x, \mathtt{b}\rangle$ sublist occupies in terms of positions *not* used by the $\langle x, \mathtt{a}\rangle$ sublist. For instance, a 1 in the $\langle x, \mathtt{b}\rangle$ sublist would represent the first position not used by an $\mathtt{a}$ within context x. Similarly, when we encode later sublists, we encode only those positions not used by the $\langle x, \mathtt{a}\rangle$ and $\langle x, \mathtt{b}\rangle$ sublists, and so on.

The ramifications of our approach in terms of search time is that the search would be sequential in terms of the number of sublists within a context, in order to decompose the relative positions that are stored. For instance, at level $k = 0$, to lookup the ith position in the $\langle x, \mathtt{c}\rangle$ sublist, we have to find the position j of the ith non-position in the $\langle x, \mathtt{b}\rangle$ sublist. Then we have to find the position of the jth non-position in the $\langle x, \mathtt{a}\rangle$ sublist. Thus, the cost of a search goes up by a factor of t_k^x for queries on context x, since we potentially have to backtrack through each sublist to resolve the query.

LEMMA 4.1. *We can remove the $O(n_k \lg e)$ space requirement at each level from the compressed suffix array, increasing the search times by a factor of at most t_k.*

4.2 The Wavelet Tree The main difficulty with the approach of Lemma 4.1 is the large sequential time cost associated with reconstructing a query to a sublist. We present instead a *wavelet tree* data structure, which is of independent interest, to reduce the query time to $\lg t_k^x \leq 2^k \lg |\Sigma|$, while still encoding in the desired space of $nH_h + o(n_k)$ bits per level. In order to prove Theorems 4.1 and 4.2, we require this technique only for level $k = 0$ (since $n_k = o(n)$ for all other k); the slowdown is at most $\lg t_0 \leq \lg |\Sigma|$.

LEMMA 4.2. (WAVELET TREE COMPRESSION) *Using a wavelet tree for each context x at level k, we can resolve a query on a sublist in $\lg t_k^x \leq \lg t_k$ time, while replacing the $O(n_k \lg e)$ term with $O(n_k \lg \lg n_k \lg t_k / \lg n_k)$ bits of space for each level k.*

Proof. It suffices to consider the representation for a single context x at some level k. All other wavelet trees are constructed similarly. For ease of presentation, let s_i be the $\langle x, i\rangle$ sublist. Let $n_k^{x,i} = |s_i|$.

We implicitly consider each left branch to be associated with a **0** and each right branch to be associated with a **1**. Each internal node is a fully-indexable dictionary [17] with the elements in its left subtree stored as **0**, and the elements in its right subtree stored as **1**. For instance, the root node of the tree in Figure 2 represents each of the positions of $s_1, \ldots, s_4$ as a **0**, and each of the positions of $s_5, \ldots, s_8$ as a **1**.

The major point here is that each internal node represents elements relative to its subtrees. Rather than the linear relative encoding of sublists we had before, we use a tree structure to exploit exponential relativity, thus reducing the length of the chain of querying significantly. In some sense, the earlier approach corresponds to a completely skewed wavelet tree, as opposed to the balanced structure now. To resolve a query for the dth entry in sublist s_i, we follow these steps:

1. Set $s = s_i$.
2. If i is odd, search for the dth **0** in s's parent.
3. If i is even, search for the dth **1** in s's parent.
4. Set $d =$ the position found in s's parent.
5. Set $i = \lfloor (i+1)/2 \rfloor$. Set $s = parent(s)$.
6. Recurse to step 2, unless $s = root$.
7. Return d as the answer to the query in sublist s_i.

This query trivially requires $\lg t_k^x$ time for context x and at most $\lg t_k$ time at level k. We analyze the space required in terms of the contribution of each internal node's fully-indexable dictionary. We prove that this cost is exactly the entropy encoding. (In some sense, we are calculating the space requirements for each s_i, propagated over the entire tree. For instance, s_i is implicitly stored in each node of its root-to-leaf path.) By induction, we can prove that the space required among all internal nodes is

$$\lg \binom{n_k^{x,1} + n_k^{x,2}}{n_k^{x,2}} + \ldots + \lg \binom{n_k^{x,t_k^x - 1} + n_k^{x,t_k^x}}{n_k^{x,t_k^x}}$$
$$+ \lg \binom{n_k^{x,1} + \ldots + n_k^{x,4}}{n_k^{x,3} + n_k^{x,4}} + \ldots + \lg \binom{n_k^{x,t_k^x - 3} + \ldots + n_k^{x,t_k^x}}{n_k^{x,t_k^x - 1} + n_k^{x,t_k^x}}$$
$$\vdots$$
$$+ \lg \binom{n_k^{x,1} + \ldots + n_k^{x,t_k^x}}{n_k^{x,1} + \ldots + n_k^{x,t_k^x/2}} = \lg \binom{n_k^x}{n_k^{x,1}, n_k^{x,2}, \ldots, n_k^{x,t_k^x}}$$

Hence, each wavelet tree encodes a particular context in precisely the empirical entropy, which is what we wanted. Summing over all contexts, we use nH_h bits per level by the definition of nH_h.

However, the sum of the $O(n_k^{x,y} \lg \lg n_k^{x,y} / \lg n_k^{x,y})$ terms at *each* of the $\lg t_k^x$ levels of the wavelet tree for some context x at level k sum to $O(n_k^x \lg \lg n_k^x / \lg n_k^x)$. Thus, we end up paying, for each context x, $O(n_k^x \lg \lg n_k^x \lg t_k^x / \lg n_k^x)$ bits, which becomes

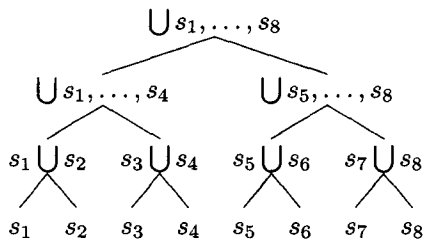

Figure 2: A wavelet tree

$O(n_k \lg\lg n_k \lg t_k / \lg n_k)$ when summed over all contexts x at level k. Thus we prove Lemma 4.2.

5 Applications to Text Indexing

In this section, we show that compressed suffix arrays are high-order entropy-compressed text indexes supporting searching of a sequence pattern P of length m in $O(m + \text{polylog}(n))$ time with only $nH_h + o(n)$ bits, where nH_h is the information theoretical upper bound on the number of bits required to encode the text T of length n. We also describe a text index that takes $o(m)$ search time *and* uses $o(n)$ bits on highly compressible texts with small alphabet size. To our knowledge, these are the first such results in text indexing.

5.1 A Pattern Matching Tool We need a search tool for a list of r sequences $S_1 \leq \cdots \leq S_r$ in lexicographical order, so that we can identify the least sequence S_i having P as a prefix in $O(m + r)$ time. (Identifying the greatest such sequence is analogous.) Our method examines the sequences $S_1, \ldots, S_r$ in left-to-right order. The steps are detailed below, where we denote the kth symbol of a sequence S by $S[k]$:

1. Set $i = 1$ and $k = 1$.
2. Find the smallest $j \geq i$ such that $S_j[k] = P[k]$.
3. If $j > r$, declare that P is not the prefix of any sequence and quit with a failure. Otherwise, assign the value of j to i, and increment k.
4. If $k \leq m$ then go to step 2. Otherwise, check whether S_i has P as a prefix, returning S_i as the least sequence in case of success or declaring a failure otherwise.

Denoting the positions assigned to i in step 3 with $i_k \geq \cdots \geq i_2 \geq i_1$, we observe that we do not access the first $k-1$ symbols of $S_{i_{k-1}+1}, \ldots, S_{i_k}$ when $i_k > i_{k-1}$, which could be potential mismatches. In general, we compare only a total of $O(i_k + k)$ symbols of $S_{i_1}, \ldots, S_{i_k}$ against those in P, where $i_k \leq r$. Only when we have reached the end of the pattern P (i.e., when $k = m$), do we increment k, set $i = i_m$, and perform a full comparison of P against S_i. This results in a correct method notwithstanding potential mismatches.

LEMMA 5.1. *Given a list of r sequences $S_1 \leq \cdots \leq S_r$ in lexicographical order, let S_i be sequence identified by our search tool. If P is a prefix of S_i, then S_i is the least sequence with this property. Otherwise, no sequence in $S_1, \ldots, S_r$ has P as a prefix. The cost of the search is $O(m + r)$ time, where m is the length of P.*

What if $S_1, \ldots, S_r$ are implicitly stored in our compressed suffix array, say at consecutive positions $x+1, \ldots, x+r$ for a suitable value of x? To achieve this goal, we need to decompress each suffix S_j on the fly by knowing its position $x+j$ in the compressed suffix array (recall that $SA[x+j]$ contains the starting position of S_j in the text). Decompressing one text symbol of S_j at a time is inherently sequential as in [2] and [19, 20]. But steps 2–3 of our search tool require us to start decompressing from the kth symbol of suffix S_j, rather than the first, which could cost us $O(mr)$ time!

Fortunately, we can overcome this problem by using the inverse compressed suffix array. In order to incrementally decompress symbols from position k of suffix S_j (having position $x + i$), we decompress the *first* symbols in the suffix at position $SA^{-1}[SA[x+i]+k-1]$ in the compressed suffix array, where SA and SA^{-1} denote the suffix array and its inverse as mentioned in Section 2. Equivalently, the latter suffix can be seen as obtained by removing the first $k-1$ symbols from S_j. All this requires a constant number of *lookup* operations and a single *substring* operation, with a cost that is independent of the value of k.

LEMMA 5.2. *Given a sequence of r consecutive suffixes $S_1 \leq \cdots \leq S_r$ in the compressed suffix array, our search tool finds the leftmost and the rightmost suffix having P as a prefix, in $O(m + r)$ symbol comparisons plus $O(r)$ lookup and substring operations, where $m = |P|$.*

5.2 High-Order Entropy-Compressed Text Indexing We now have all the tools to describe our search of P in the compressed suffix array. We first perform a search of P in $SA_{\ell+\lg t(n)}$, which is stored explicitly along with $LCP_{\ell+\lg t(n)}$, the longest common prefix information required in [10]. (The term $t(n)$ depends on which implementation of compressed suffix arrays we use.) We require $O(m)$ symbol comparisons plus $O(\lg n)$ *lookup* and *substring* operations. At that point, we locate a portion of the (compressed) suffix array storing $r = 2^{\ell+\lg t(n)} = O(t(n) \lg n)$ suffixes. We run our search tool on these r suffixes, at the cost of $O(m + t(n) \lg n)$ symbol comparisons and $O(t(n) \lg n)$ calls to *lookup* and *substring*, which is also the asymptotical cost of the whole search.

THEOREM 5.1. *Given a text of n symbols over the alphabet Σ, we can replace it by a compressed suffix*

array occupying $\epsilon^{-1}nH_h + O(n \lg\lg n/\lg^{\epsilon}_{|\Sigma|} n)$ *bits, so that searching a pattern of length* m *takes* $O(m/\lg_{|\Sigma|} n + (\lg n)^{(1+\epsilon)/(1-\epsilon)}(\lg|\Sigma|)^{(1-3\epsilon)/(1-\epsilon)})$ *time, for any fixed value of* $0 < \epsilon < 1/2$.

For example, fixing $\epsilon = 1/3$ in Theorem 5.1, we obtain a search time of $O(m/\lg_{|\Sigma|} n + \lg^2 n)$ with a self-index occupying $3nH_h + O(n \lg\lg n/\lg^{1/3}_{|\Sigma|} n)$ bits. We can reduce the space to nH_h bits plus a lower-order term, obtaining a nearly space-optimal self-index.

THEOREM 5.2. *Given a text of* n *symbols over the alphabet* Σ, *we can replace it by a compressed suffix array occupying nearly optimal space, i.e.,* $nH_h + O(n \lg\lg n/\lg_{|\Sigma|} n)$ *bits, so that searching a pattern of length* m *takes* $O(m \lg|\Sigma| + \lg^4 n/\lg^2 \lg n \lg|\Sigma|) = O(m \lg|\Sigma| + \text{polylog}(n))$ *time.*

If we augment the compressed suffix array to obtain the hybrid multilevel data structure in [6], we can improve the lower-order terms in the search time of Theorem 5.1, where $t(n) = \lg^{\gamma}_{|\Sigma|} n$ and $\gamma = \epsilon/(1-\epsilon) > \epsilon$. We use a sparse suffix tree storing every other $(t(n) \lg n)$th suffix using $O(n/t(n)) = O(n/\lg^{\epsilon}_{|\Sigma|} n)$ bits to locate a portion of the (compressed) suffix array storing $O(t(n) \lg n)$ suffixes. However, we do not immediately run our search tool in Lemma 5.2; instead, we employ a nested sequence of space-efficient Patricias [12] of size $\lg^{\omega-\gamma} n$ until we are left with segments of $r = t(n)$ adjacent suffixes in the compressed suffix array, for any fixed value of $1 > \omega \geq 2\gamma > 0$. This scheme adds $O(n/r) = O(n/\lg^{\epsilon}_{|\Sigma|} n)$ bits to the self-index, allowing us to restrict the search of pattern P to a segment of r consecutive suffixes in the compressed suffix array. At this point, we run our search tool in Lemma 5.2 on these r suffixes to identify the leftmost occurrence of the pattern.

THEOREM 5.3. *Given a text of* n *symbols over the alphabet* Σ, *we can replace it by a hybrid compressed suffix array occupying* $\epsilon^{-1}nH_h + O(n \lg\lg n/\lg^{\epsilon}_{|\Sigma|} n)$ *bits, so that searching a pattern of length* m *takes* $O(m/\lg_{|\Sigma|} n + \lg^{\omega} n \log^{1-\epsilon}|\Sigma|)$ *time, for any fixed value of* $1 > \omega \geq 2\epsilon/(1-\epsilon) > 0$ *and* $0 < \epsilon < 1/3$.

We provide the first self-index with small alphabets that is sublinear both in space and in search time.

COROLLARY 5.1. *For any text where* $nH_h = o(n)$ *and the alphabet is small, the self-index in Theorem 5.3 occupies just* $o(n)$ *bits and requires* $o(m)$ *search time.*

Acknowledgments

We would like to thank Rajeev Raman, Venkatesh Raman, S. Srinivasa Rao, and Kunihiko Sadakane for sending us a copy of the full journal version of their papers, and Rajeev Raman and S. Srinivasa Rao for clarifying some details on succinct data structures.

References

[1] D. E. Ferguson. Bit-Tree: a data structure for fast file processing. *C. ACM*, 35(6):114–120, 1992.

[2] P. Ferragina, G. Manzini. Opportunistic data structures with applications. In *FOCS*, 390–398, 2000.

[3] P. Ferragina, G. Manzini. An experimental study of an opportunistic index. In *SODA*, 269–278, 2001.

[4] G. H. Gonnet, R. A. Baeza-Yates, T. Snider. New indices for text: PAT trees and PAT arrays. In *Information Retrieval: Data Struct. Algs.*, 66–82, 1992.

[5] `http://www.google.com/help/refinesearch.html`.

[6] R. Grossi, J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *STOC*, 397–406, 2000.

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

[8] T. Luczak, W. Szpankowski. A Suboptimal Lossy Data Compression Based in Approximate Pattern Matching. *IEEE Trans. Inform. Theory*, 43, 1439–1451, 1997.

[9] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[10] U. Manber, G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22:935–948, 1993.

[11] D. R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded In Alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[12] J. I. Munro, V. Raman, S. S. Rao. Space efficient suffix trees. *J. Algorithms*, 39:205–222, 2001.

[13] J. I. Munro. Tables. *FSTTCS*, 16:37–42, 1996.

[14] A. Moffat, J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Informat. Systems*, 14(4):349–379, 1996.

[15] G. Navarro, E. Silva de Moura, M. Neubert, N. Ziviani, R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Inform. Retrieval*, 3:49–77, 2000.

[16] R. Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J.Comput.*, 31:353–363, 2001.

[17] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *SODA*, 233–242, 2002.

[18] S. S. Rao. Time-space trade-offs for compressed suffix arrays. *IPL*, 82(6):307–311, 2002.

[19] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *ISAAC*, 410–421, 2000.

[20] K. Sadakane. Succinct representations of *lcp* information and improvements in the compressed suffix arrays. In *SODA*, 2002.

[21] I. H. Witten, A. Moffat, T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann, second edition, 1999.