SUCCINCT DATA STRUCTURES

by

Ankur Gupta

Department of Computer Science
Duke University

Date: ____________________

Approved:

____
Jeffrey Scott Vitter, Supervisor

____
Pankaj Agarwal

____
Roberto Grossi

____
Xiaobai Sun

____

Dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

2007

ABSTRACT

SUCCINCT DATA STRUCTURES

by

Ankur Gupta

Department of Computer Science
Duke University

Date: ______________________________

Approved:

____
Jeffrey Scott Vitter, Supervisor

____
Pankaj Agarwal

____
Roberto Grossi

____
Xiaobai Sun

____

An abstract of a dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

2007

Abstract

The world is drowning in data. The recent explosion of web publishing, XML data, bioinformation, scientific data, image data, geographical map data, and even email communications has put a strain on our ability to manage the information contained there. The influx of massive data sets with all kinds of features presents a number of difficulties with efficient management of storage space, organization of information, and data accessibility. A primary computing challenge in these cases is how to compress the data but still allow them to be queried quickly. This thesis addresses theoretical and algorithmic issues arising from these practical concerns for the problem of *compressed text indexing*, where we want to maintain efficient data storage and rapid response to queries on data.

The premise of data compression comes from many real-life situations, where data are often highly compressible. This compressibility constitutes a major opportunity for saving space and data query latency, and is a critical bottleneck for many applications. In mobile applications, for instance, space and the power to access information are at a premium. In a streaming environment, where new data are being generated constantly, compression can also aid in prediction of upcoming trends. In the case of bioinformatics, analyzing succinct representations of DNA sequences could lead to a deeper understanding of nature, perhaps even giving hints on secondary and tertiary structure, gene evolution, and other important topics.

We use text data as the subject of this particular study. We introduce a number of compressed data structures for compressed text indexing that enable arbitrary searching for patterns in the provably best possible time. The methodology is distinct in that the process of searching also encompasses decoding; therefore, the original document is no longer needed. Together, these data structures can be used at multiple levels of a compression-retrieval hierarchy to arrive at an overall text indexing solution. Some structures can be used individually as well, within or beyond the scope of text indexing. For each data structure, we provide a theoretical estimate of its space usage and query performance on a suite of operations crucial to access the stored data. In each case, we relate its space usage to the *compressed size of the original data* and show that the supported operations function in near-optimal or optimal time.

We also present a number of experimental results using our methodology. These experiments validate our theoretical findings, and we establish that our methodology is competitive with the state-of-the-art.

Acknowledgements

First and foremost, I would like to thank my advisor Jeffrey Scott Vitter. I'm not sure where I would be without his continued support and guidance. Jeff's insistence on clarity and precision is a necessary foundation for any serious graduate student, and I am grateful to have benefited from such a firm vision.

I would also like to thank my committee members Roberto Grossi, Xiaobai Sun, and Pankaj Agarwal for providing careful comments on my doctoral work. Special thanks go to Roberto Grossi, who served as a collaborator and co-advisor throughout my graduate career and helped shape who I have become. I would also like to thank Rahul Shah and Wing-Kai Hon, both of with whom I enjoyed working and socializing.

I would like to thank my family for providing love and encouragement. My parents, Umesh and Manju Gupta, and my brother Parag Gupta, were always there when I most needed someone. I could not have completed this work without them. I cannot begin to express in words the impact my wife Diksha had on me during the final stages of my studies; her concern for and patience with long hours and demanding schedules are truly amazing. Finally, my grandfather Ramswaroop Gupta has always been a quiet strength in my life, with a deep calm and a focus on the simple things. I hope one day to reach that pedestal.

I have a long list of friends whose companionship has broadened my life: Matt Taylor, Rex Robinson, Sharlotte Greer, Tylan Watts, Andrew Strack, Priya Mahadevan, Justin Moore, Kristina Killgrove, Patrick Reynolds, David Cherryholmes, and Aaron Miller to name just a few. I am glad to have met them.

I would like to offer thanks to Michael E. Durbin, who advised me while I was at the University of Texas at Dallas. His mentorship played a big part in fueling my enthusiasm towards Computer Science. I would also like to thank Diane Riggs, in the

Department of Computer Science at Duke University. She was always there to offer help to students, whether it be paperwork, scheduling, or just a sympathetic ear.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The problem of data proliferation is challenging our ability to manage information. Classic algorithms are greedy in terms of their space usage and cannot access only a tiny portion of the data. This trend has not gone unnoticed by researchers, as evidenced by the recent issues in data streaming [Mut03] and sublinear algorithms [Cha04]. Unlike these cases, many problems require the entire dataset to be stored in compressed format but still need it to be queried quickly. In fact, compression may have a more far-reaching impact than simply storing data succinctly: "That which we can compress we can understand, and that which we can understand we can predict," as observed in [Aar05]. Much of what we call "insight" or "intelligence" can be thought of as simply finding succinct representations of sensory data [Bau04]. For instance, we are far from fully understanding the intrinsic structure of biological sequences, and as of today, we cannot compress them well either.

Researchers have considered these issues in several algorithmic contexts, such as the design of efficient algorithms for managing highly-compressible data structures. They have carefully studied the exact resources needed to represent trees [BDM$^+$05, GRR04, MRS01a, MRS01b, MR02], graphs [Jac89a, BBK03], sets and dictionaries [BB04, BM99, Pag01, RR03, RRR02], permutations and functions [MRRR03, MR04], and text indexing structures [FM05, GV05, GGV04, FGGV04, Sad02b, Sad03]. The goal is to design algorithms with tight space complexity $s(n)$. The Kolmogorov complexity for representing data provides a lower bound on the value of $s(n)$ for each representation studied. Kolmogorov complexity essentially defines compression in terms of the size of the smallest program that can generate the input provided [LV97]. However, Kolmogorov complexity is undecidable for arbitrary data, so any compression method is known to be suboptimal in this sense.[1]

[1]Extrapolating from [Aar05, Bau04], the undecidability of Kolmogorov complexity implies that there is a computational limit on finding succinct representations for sensory data.

The hope is to achieve $s(n) + o(s(n))$ bits, with nearly-optimal asymptotic time bounds, i.e. $O(t(n))$ time, while remaining competitive with state-of-the-art (uncompressed) data structures [Jac89a].

Providing an accurate analysis of space occupancy (up to lower-order terms) is motivated by the above theoretical issues as well as the following technological issues. Space savings can translate into faster processing (by reducing disk accesses), which results in shorter seek times or allows data storage on faster cache levels. A recent line of research uses the I/O computation model [Vit01] to take into account some of these issues, such as cache-oblivious algorithms and data structures [AV88, BDFC05]. Some algorithms exploit data compression to achieve provably better time bounds [RC93, KV98, VK96]. From an economical standpoint, compressed data would require less media to store (such as RAM chips in search engines or portable computing devices) or less time to transmit over regulated bandwidth models (such as transmissions by cell phones).

Similar goals for analyzing time bounds are difficult to achieve due to the complexity of modern machines, unless some simple computation model (such as one reminiscent of the comparison model) is used. Sources of imprecision include cache hits/misses, dynamic re-ordering of instructions to maximize instruction parallelism, disk scheduling issues, and latency of disk head movements. Space bounds, on the other hand, are relatively easier to predict and can often be validated experimentally. This concrete verification is an important component of research due to technological advances that may affect an otherwise good bound: 64-bit CPUs are on the market (increasing the pointer size or address space), Unicode text is becoming more commonplace (requiring more than 8 bits per symbol as in ASCII text), and XML databases are encoding more data as well (adding a non-trivial amount of formatting data to the "real" information). We need to squeeze all this data and provide fast access to its compressed format. For a variety of data structures, therefore, the question remains: Can we achieve a near-optimum compression and *simultaneously* support asymptotically fast queries?

In this thesis, we address this question for a number of applications focused around the problem of *compressed text indexing*. The goal is to develop an index for an input text T that can efficiently search for any arbitrary substring of the text, *and* the index itself requires space *proportional to the size of the optimally-compressed input text T*. Our work focuses on developing both the text indexes, and sheds light on the critical components necessary to achieve the best possible index. We also develop a number of these components, which are meaningful results in their own right. We now briefly overview these components, and explain how they work together.

1.1 Text Compression and Text Indexing

Our main interest is on text data. Properly addressing the text data issue also requires efficient solutions to a number of derivative succinct indexing problems. In this context, the tight space complexity $s(n)$ is better expressed in terms of the *entropy* of the particular text at hand. See [Sha48] for the definition of entropy and [CT91] for the relation between entropy and Kolmogorov complexity.

We want to develop tight space bounds for *text compression*, i.e. storing a text in a compressed binary format. We additionally want to design *compressed text indexes* to decode any small portion of the text or search for any pattern as a substring of the text, without decompressing the binary format entirely. In particular, we study how to obtain a compressed representation of the text that is a *self-index*, namely, we desire a compressed binary format that is also an index for the text itself.

We consider the text T as a sequence of n symbols, where each symbol is drawn from the alphabet Σ of size σ. Since the raw text T occupies $n \lg \sigma$ bits of storage, T is compressible if it can be represented in fewer than $n \lg \sigma$ bits.[2] It is a simple fact that no encoding of T can take fewer bits than the entropy of T, which measures how much randomness is in T. Here, entropy is related to the size of the smallest program which generates T, according

[2] In this thesis, we use the notation $\lg_b^c a = (\lg_b a)^c = (\lg a / \lg b)^c$ to denote the cth power of the base-b logarithm of a. If no base is specified, the implied base is 2.

to the Kolmogorov complexity. So, we expect that the entropy of T is a lower bound to the space complexity $s(n)$ for compressed data structures that store T.

The entropy bound is ideal, but we can only quantitatively analyze an approximation of it, namely,

$$nH_h + M(T, \Sigma, h) \tag{1.1}$$

in terms of bits of space. In formula (1.1), $H_h \leq \lg \sigma$ is the *hth-order empirical entropy* of T, which captures the dependence of symbols on their *context*, made up of the h adjacent symbols in the text T. As n increases, $M(T, \Sigma, h)$ denotes the number of bits used to store the empirical probabilities for the corresponding statistical model in T: informally, $M(T, \Sigma, h)$ represents the number of bits required to store the number of occurrences of yx as a substring of the text T, for each context x of length h and each symbol $y \in \Sigma$. (These quantities are discussed formally in Sections 2.2 and 2.3.) As h increases, nH_h is non-increasing and $M(T, \Sigma, h)$ is non-decreasing. Thus, carefully tuning the context length h gives the best choice for minimizing space. An interesting problem is how to obtain nearly optimal space bounds where $s(n)$ is approximated by formula (1.1) for the best choice of h. In practice, English text is often compressible by a factor of 3 or 4, and the best choice for h is usually about 4 or 5. Lempel and Ziv have provided an encoding such that $h \leq \alpha \lg n + O(1)$ (where $0 < \alpha < 1$) is sufficiently good for approximating the entropy; Luczak and Szpankowski prove a sufficient approximation for ergodic sources when $h = O(\lg n)$ in [LS97].

In Chapter 2, we present a unified algorithmic framework to obtain nearly optimal space bounds for text compression and compressed text indexing, apart from lower-order terms. In particular, we provide a tight analysis of the Burrows-Wheeler transform (BWT) establishing a bound of $nH_h + M(T, \Sigma, h)$ bits Using the same framework, we also obtain an implementation of the compressed suffix array (CSA) that achieves $nH_h + M(T, \Sigma, h) + O(n \lg \lg n / \lg_{|\Sigma|} n)$ bits of space while still retaining competitive full-text indexing functionality.

The novelty of the proposed framework lies in its use of the finite set model instead of the empirical probability model (as in previous work), giving us new insight into the design and analysis of our algorithms. For example, we show that our analysis gives improved bounds since $M(T, \Sigma, h) \leq \min\{g'_h \lg(n/g'_h + 1), H^*_h n + \lg n + g''_h\}$, where $g'_h = O(|\Sigma|^{h+1})$ and $g''_h = O(|\Sigma|^{h+1} \lg |\Sigma|^{h+1})$ do not depend on the text length n, while $H^*_h \geq H_h$ is the modified hth-order empirical entropy of T. We go on to describe some classes of texts for which the above bound is nearly tight, showing that they are among the hardest to compress with the BWT. We also examine the importance of lower-order terms, as these can dwarf any savings achieved by high-order entropy. Moreover, we show a strong relationship between a compressed full-text index and the succinct dictionary problem. This last consequence is a key observation, since it neatly separates the text indexing problem into that of encoding a series of dictionary data structures.

In Chapter 3, we also report on a new experimental analysis of high-order entropy-compressed suffix arrays, which retains the theoretical performance of previous work and represents an improvement in practice. Our experiments indicate that the resulting text index offers state-of-the-art compression. In particular, we require roughly 20% of the original text size—without requiring a separate instance of the text. We can additionally use a simple notion to encode and decode block-sorting transforms (such as the Burrows-Wheeler transform), achieving a compression ratio comparable to that of `bzip2`. We also provide a compressed representation of suffix trees (and their associated text) in a total space that is comparable to that of the text alone compressed with `gzip`.

1.2 Dictionaries and Data-Aware Measures For Set Data

In Chapter 4, we consider the fundamental *dictionary problem* on *set data*, where the task is to construct a data structure for representing a set S of n items out of a universe $U = \{0, \ldots, u-1\}$ and supporting various queries on S. Dictionaries are used extensively in text indexing and other applications with a text input (such as the database applications of XML selectivity estimation) as a building block in designing entropy-compressed data structures. For text-based applications, dictionaries serve as a powerful black box that operate within some entropy-aware partitioning of the data. Any improvements to a dictionary structure would have tremendous impact on all such dependent applications.

We use a well-known data-aware measure for set data called *gap* to bound the space of our data structures. We describe a novel dictionary structure that requires $gap + O(n \lg(u/n)/\lg n) + O(n \lg\lg(u/n))$ bits. Under the RAM model, our dictionary supports membership, rank, and predecessor queries in nearly optimal time, matching the time bound of Andersson and Thorup's predecessor structure [AT00], while simultaneously improving upon their space usage. We support select queries even faster in $O(\lg\lg n)$ time.

Our dictionary structure uses exactly *gap* bits in the leading term (i.e., the constant factor is 1) and answers queries in near-optimal time. When seen from the worst case perspective, we present the first $O(n \lg(u/n))$-bit dictionary structure that supports these queries in near-optimal time under the RAM model. We also build a dictionary that requires the same space and supports membership, select, and partial rank queries even more quickly in $O(\lg\lg n)$ time.

We show that for many (real-world) datasets, data-aware methods lead to a worthwhile compression over combinatorial methods. To our knowledge, these are the first results that achieve data-aware space usage and retain near-optimal time.

1.3 Dynamizing Succinct Data Structures

We present a framework in Chapter 5 to dynamize succinct data structures, to encourage their use over non-succinct versions in a wide variety of important application areas. Our framework can dynamize most state-of-the-art succinct data structures for dictionaries, ordinal trees, labeled trees, and text collections. Of particular note is its direct application to XML indexing structures that answer *subpath* queries [FLMM05]. Our framework focuses on achieving information-theoretically optimal space along with near-optimal update/query bounds.

As the main part of our work, we consider the following problem central to text indexing: Given a text T over an alphabet Σ, construct a compressed data structure answering the queries $\mathit{char}(i)$, $\mathit{rank}_s(i)$, and $\mathit{select}_s(i)$ for a symbol $s \in \Sigma$. Many data structures consider these queries for static text T [GGV03, FM05, SG06, GMR06]. We build on these results and give the best known query bounds for the dynamic version of this problem, supporting arbitrary insertions and deletions of symbols in T.

Specifically, with an amortized update time of $O(n^{\epsilon})$, any static succinct data structure D for T, taking $t(n)$ time for queries, can be converted by our framework into a dynamic succinct data structure that supports $\mathit{rank}_s(i)$, $\mathit{select}_s(i)$, and $\mathit{char}(i)$ queries in $O(t(n) + \lg\lg n)$ time, for any constant $\epsilon > 0$. When $|\Sigma| = \mathrm{polylg}(n)$, we achieve $O(1)$ query times. Our update/query bounds are near-optimal with respect to the lower bounds from [PD06].

The best previously-known query times for this problem were $O(\lg n \lg |\Sigma|)$, given by [NM06b], although their update bounds are also $O(\lg n \lg |\Sigma|)$. Our framework can be easily modified to achieve similar bounds.

Nevertheless, we focus on faster query/slower update for both theoretical and practical considerations. Theoretically speaking, our query bounds match (or nearly match) the bounds given by fastest known static data structures. With this query bounds as the target, our update bounds are nearly tight with respect to the applicable lower bounds known for the partial sums problem [PD06]. Practically, our choice of faster query/slower update is well-suited for many data structuring environments in string matching, databases and XML indexing.

Chapter 2

An Algorithmic Framework for Compression and Text Indexing

2.1 Introduction

In this chapter, we describe a *unified algorithmic framework* that achieves the first *nearly optimal* space bounds for *both* text compression and compressed text indexing. We provide a new tight analysis of text compression based on the *Burrows-Wheeler transform* [BW94] (hereafter called the BWT). We also provide a new implementation of compressed text indexing based on the *compressed suffix array* [FM05, GV05, Sad03] (hereafter called the CSA). A key point of our unified approach is the use of the *finite set model* instead of the empirical probability model adopted in previous work, giving us new insight into the analysis. We capture the empirical probabilities encoded in $M(T, \Sigma, h)$ bits (see Formula (1.1)) by employing a two-dimensional conceptual organization which groups contexts x from the text by their predicted symbols y. This scheme can be seen as an alternative way to model an arbitrary partition of the BWT. We then restructure each context accordingly, encoding each group with an algorithm that stores t items out of a universe of size n in the information theoretic minimum space $\lceil \lg \binom{n}{t} \rceil$ bits (since there are $\binom{n}{t}$ subsets of t items out of n). In Sections 2.1.1 and 2.1.2, we detail our results for text compression and text indexing, which reach nearly optimal space bounds for both areas. The work in this chapter was a collaborative effort with Roberto Grossi and Jeffrey Scott Vitter.

2.1.1 Text Compression

In this section, we discuss our results for text compression, which are based on the Burrows-Wheeler transform (BWT). Simply put, the BWT rearranges the text T so that it is easily compressed by other methods. In practice, the compressed version of this transformed text is quite competitive with other methods [Fen96, Fen02, FTL03]. The BWT is at the heart of compressors based on block-sorting (such as `bzip2`) that outperform Lempel-Ziv-based compressors (such as `gzip`). We provide a method for representing the BWT in compressed format using well-known results from combinatorial enumeration [Knu05, Rus05] in an unusual way, exploiting the functionality of ranking and unranking t-subsets for compressing and decompressing the t items thus stored.[1] We collect and store this information in our new *wavelet tree*, a novel data structure that we use to represent the LF mapping (from the BWT and used in the FM-index [FM05]) and the neighbor function Φ (at the heart of the CSA [GV05]). Our framework-based analysis gives a bound of $nH_h + M(T, \Sigma, h)$ bits for any given h as input, thus matching Formula (1.1). The best value of h can be found using the optimal partitioning of the BWT as given in [FGMS05], so that our bound holds for any h (simply because Formula (1.1) cannot be smaller for the other values of h). For comparison purposes, we give an upper bound on the number of bits needed to encode the statistical model,

$$M(T, \Sigma, h) \leq \min\left\{g'_h \lg(n/g'_h + 1),\ H^*_h n + \lg n + g''_h\right\}, \tag{2.1}$$

where $g'_h = O(\sigma^{h+1})$ and $g''_h = O(\sigma^{h+1} \lg \sigma^{h+1})$ do not depend on the text length n.

In Formula (2.1), $H^*_h \geq H_h$ is the *modified* hth-order empirical entropy (see Section 2.2) introduced in [Man01] to show that the BWT can be represented in at most $(5+\epsilon)nH^*_h + \lg n + g_h$ bits, where $\epsilon \approx 10^{-2}$ and $g_h = O(\sigma^{h+1} \lg \sigma)$. The latter bound is important for low-entropy texts, since the compression ratio scales with high-order entropy; this bound cannot be attained when replacing H^*_h by H_h. We refer the reader to [Man01] for previous literature on the subject. In contrast, the compression ratio of Lempel-Ziv algorithm [ZL77] does not scale for low-entropy texts: although its output is bounded by $nH_h + O(n \lg\lg n / \lg n)$ bits, it cannot be smaller than $2.5nH_0$ bits for some strings [KM99]. Note that the BWT is a booster for 0th-order compressors as shown in [FGMS05], where a close connection of the

[1]We use the term t-subset instead of the more usual k-subset terminology, because we use k to denote the levels of our compressed suffix array (described later). A similar observation holds for entropy H_h, which is often referred to as H_k in the literature.

optimal partition of the BWT with the suffix tree [McC76] attains the best known space bounds for the analysis of the BWT, namely, $2.5nH^*_h + \lg n + g_h$ bits and $nH_h + n + \lg n + g_h$ bits. The related compression methods do not require the knowledge of the order h and take $O(n \lg \sigma)$ time for general alphabets.

Using (2.1), we can compare our analysis with the best bounds from previous work. When compared to the additive term of $O(n \lg \lg n / \lg n)$ in the analysis of the Lempel-Ziv method in [KM99], we obtain an $O(\lg n)$ additive term for $\sigma = O(1)$ and $h = O(1)$, giving strong evidence why the BWT is better than the Lempel-Ziv method. Indeed, since $M(T, \Sigma, h) \leq g'_h \lg(n/g'_h + 1)$, our bound in (1.1) becomes $nH_h + O(\lg n)$ when $h = O(1)$ and $\sigma = O(1)$, thus exponentially reducing the additive term of n of the H_h-based analysis in [FGMS05]. In this case, our bound closes the gap in the analysis of BWT, since it matches the lower bound of $nH_h + \Omega(\lg \lg n)$, up to lower-order terms. The latter comes from the lower bound of $nH^*_0 + \Omega(\lg \lg n)$ bits, holding for a large family of compressors (not necessarily related to BWT), as shown in [FGMS05]; the only (reasonable) requirement is that any such compressor must produce a codeword for the text length n when it is fed with an input text consisting of the same symbol repeated n times. Since $H_h \leq H^*_0$, we easily derive the lower bound of $nH_h + \Omega(\lg \lg n)$ bits, but a lower bound of $nH_h + \Omega(\lg n)$ probably exists since $nH^*_0 \geq \lg n$ while nH_h can be zero.

As for the modified hth-order empirical entropy, we show that our analysis in (1.1) can be upper bounded by $n(H_h + H^*_h) + \lg n + g''_h$ bits using (2.1). Since $H_h \leq H^*_h$, our bound in (1.1) is strictly smaller than $2.5nH^*_h + \lg n + g_h$ bits in [FGMS05], apart from the lower-order terms. Actually, our bound is definitively smaller in some cases. For example, while a bound of the form $nH^*_h + \lg n + g_h$ bits is not always possible [Man01], there are an infinite number of texts for which $nH_h = 0$ while $nH^*_h \neq 0$. In these cases, our bound from (1.1) is $nH^*_h + \lg n + g''_h$ bits.

We also describe a class of non-trivial texts where our bound is nearly tight. In particular, we show that our analysis is nearly tight for any chosen positive constant $0 < \delta \leq 1$, namely, there exists an infinite family of strings such that for any n-long string in the family, its bound in Formula (1.1) satisfies $n(((k-1)/k)\delta H_h + H_h^*) - o(nH_h^*) \leq nH_h + M(T, \Sigma, h) \leq n(\delta H_h + H_h^*) + \lg n + g_h''$, where $k > \lceil 1/\delta \rceil$ is a constant. (The definition of these families is intimately related to our analysis.) Finally, encoding and decoding take $O\big(n(nH_h/\lg n + 1) + g_h''\big)$ time; however, as shown in [GGV04], we can use run-length encoding in place of subset encoding in a practical setting, reducing the time complexity to $O(n \lg \sigma)$.

2.1.2 Compressed Text Indexing

In this section, we discuss our analysis with respect to text indexing based on the compressed suffix array (CSA). Text indexing data structures preprocess a text T of n symbols drawn from an alphabet Σ such that any query pattern P of m symbols can be answered quickly without requiring an entire scan of the text itself. We denote a substring $T[i]T[i+1]\cdots T[j]$ of contiguous text symbols by $T[i,j]$. Depending on the type of query, we may want to know if P occurs in T (occurrence or search query), how many times P occurs in T (counting query), or the locations where P occurs in T (enumerative query). An occurrence of pattern P at position i identifies a substring $T[i, i+m-1]$ equal to P. Because a text index is a preprocessed structure, a reasonable query time should have no more than a polylg(n) cost plus an output sensitive cost $O(occ)$, where occ is the number of occurrences retrieved (which is crucial for large-scale processing).

Until recently, these data structures were greedy of space and also required a separate (original) copy of the text to be stored. Suffix trees [McC76, Ukk95, Wei73] and suffix arrays [GBS92, MM93] are prominent examples. The suffix tree is a compact trie whose leaves store each of the n suffixes contained in the text T, namely, $T[1,n], T[2,n], \ldots, T[n,n]$, where suffix $T[i,n]$ is uniquely identified by its starting position i. Suffix trees [McC76, MM93] allow fast queries of substrings (or patterns) in T in $O(m \lg \sigma + occ)$ time, but require at least $4n \lg n$ bits of space, in addition to keeping the text. The suffix array SA is another popular index structure. It maintains the permuted order of $1, 2, \ldots, n$ that corresponds to the locations of the suffixes of the text in lexicographically sorted order,

$T\big[SA[1],n\big]$, $T\big[SA[2],n\big]$, $\ldots$, $T\big[SA[n],n\big]$. Suffix arrays [GBS92, MM93] (that store the length of the longest common prefix) are nearly as good at searching as are suffix trees. Their time for finding occurrences is $O(m + \lg n + \mathit{occ})$ time, but the space cost is at least $n \lg n$ bits, plus the cost of keeping the text.

A new trend in the design of modern indexes for full-text searching is addressed by the CSA [GV05, Rao02, Sad03, Sad02b] and the opportunistic FM-index [FM05], the latter making the very strong intuitive connection between the power of the BWT and suffix arrays. They support the functionalities of suffix arrays and overcome the aforementioned space limitations. In our framework, we implement the CSA by replacing the basic t-subset encoding with succinct dictionaries supporting constant-time *rank* and *select* queries. The *rank* query returns the number of entries in the dictionary that are less than or equal to the input entry; the *select* query returns the ith entry in the dictionary for the input i. Succinct dictionaries store t keys over a bounded universe n in the information theoretically minimum space $\lceil \lg \binom{n}{t} \rceil$ bits, plus lower-order terms $O(n \lg\lg n / \lg n) = o(n)$ [RRR02]. We show a close relationship between compressing a full-text index with high-order entropy to the succinct dictionary problem. Prior to the work of this chapter, the best space bound was $5nH_h + O\big(n\frac{\sigma + \lg\lg n}{\lg n} + n^{\epsilon}\sigma 2^{\sigma \lg \sigma}\big)$ bits for the FM-index, supporting a new backward search algorithm in $O(m + \mathit{occ} \times \lg^{1+\epsilon} n)$ time for any $\epsilon > 0$ [FM05]. We refer the reader to the survey in [NM06a] for a discussion of more recent work in this area.

We obtain several tradeoffs between time and space as shown in Tables 2.1 and 2.2. For example, Theorem 13 gives a self-index requiring $nH_h + O(n \lg\lg n / \lg_\sigma n)$ bits of space (where $h + 1 \leq \alpha \lg_\sigma n$ for an arbitrary positive constant $\alpha < 1$) that allows searching for patterns of length m in $O(m \lg \sigma + \mathit{occ} \times \mathrm{polylg}(n))$ time. Thus, using our new analysis of the BWT, our implementation provides the first self-index reaching the high-order empirical entropy nH_h of the text with a multiplicative constant of 1; moreover, we conjecture that $g'_h \lg(n/g'_h + 1)$ additional bits are not achievable for text indexing. If true, this claim would imply that adding self-indexing capabilities to a compressed text requires more space than $M(T, \Sigma, h)$, the number of bits encoding

bits of space	*lookup* & *lookup*$^{-1}$	*substring*	conditions	notes
$nH_h \lg\lg_\sigma n + o(n\lg\sigma) + O\big(\sigma^h(n^\beta+\sigma)\big)$	$O(\lg\lg_\sigma n)$	$O(\frac{c}{\lg_\sigma n} + \lg\lg_\sigma n)$	any $0<\beta<1$	Thm.9
$\epsilon^{-1} nH_h + O\big(\frac{n\lg\lg n}{\lg^\epsilon_\sigma n} + \sigma^h(n^\beta+\sigma)\big)$	$O\big((\lg_\sigma n)^{\epsilon/1-\epsilon}\lg\sigma\big)$	$O\big(\frac{c}{\lg_\sigma n} + (\lg_\sigma n)^{\epsilon/1-\epsilon}\lg\sigma\big)$	any $0<\beta<1$, $0<\epsilon\le 1/2$	Thm.10
$\epsilon^{-1}nH_h + O(n) + O\big(\sigma^h(n^\beta+\sigma)\big)$	$O\big((\lg_\sigma n)^{\epsilon/1-\epsilon}\big)$	$O\big(\frac{c}{\lg_\sigma n} + (\lg_\sigma n)^{\epsilon/1-\epsilon}\big)$	$n = o(n\lg\sigma)$ for $\sigma=\omega(1)$	Cor.3
$nH_h + O\big(\frac{n\lg\lg n}{\lg_\sigma n} + \sigma^{h+1}\lg(1+n/\sigma^{h+1})\big)$	$O(\lg^2 n/\lg\lg n)$	$O(c\lg\sigma + \lg^2 n/\lg\lg n)$	any $0<\beta<1$	Thm.11

Table 2.1: Trade-offs between time and space for the implementation of CSA and its supported operations. (See Definition 2.) The lower-order terms in the space complexity are all $o(n\lg\sigma)$ bits except $\sigma^h(n^\beta+\sigma)$ (because of $M(T,\Sigma,h)$), which is $o(n\lg\sigma)$ when $h+1\le\alpha\lg_\sigma n$ for any arbitrary positive constant $\alpha<1$ (we fix β such that $\alpha+\beta<1$). In all cases, *compress* requires $O(n\lg\sigma+\sigma^h(n^\beta+\sigma))$ time.

bits of space	search/count time	enumerative time (per item)	conditions	notes
$\epsilon^{-1} nH_h + O(\frac{n \lg\lg n}{\lg_\sigma^\epsilon n})$	$O\big(\frac{m}{\lg_\sigma n} + (\lg n)^{(1+\epsilon)/(1-\epsilon)} (\lg \sigma)^{(1-3\epsilon)/(1-\epsilon)}\big)$	$O\big((\lg n)^{(1+\epsilon)/(1-\epsilon)} (\lg \sigma)^{(1-3\epsilon)/(1-\epsilon)}\big)$	any $0 < \epsilon \leq 1/2$	Thm.12
$nH_h + O(\frac{n \lg\lg n}{\lg_\sigma n})$	$O(m \lg \sigma + \lg^4 n / (\lg^2 \lg n \lg \sigma))$	$O(\lg^4 n / (\lg^2 \lg n \lg \sigma))$	$1 > \omega \geq 2\epsilon/(1-\epsilon)$	Thm.13
$\epsilon^{-1} nH_h + O(\frac{n \lg\lg n}{\lg_\sigma^\epsilon n})$	$O(\frac{m}{\lg_\sigma n} + \lg^\omega n \lg^{1-\epsilon} \sigma)$	$O(\lg^\omega n \lg^{1-\epsilon} \sigma)$	$0 < \epsilon \leq 1/3$	Thm.14

Table 2.2: Trade-offs between time and space for the compressed text indexing based on the CSA, under the assumption that $h + 1 \leq \alpha \lg_\sigma n$ for any arbitrary positive constant $\alpha < 1$. The lower-order terms in the space complexity are all $o(n \lg \sigma)$ bits. In all cases, the construction takes $O(n \lg \sigma)$ time and uses a temporary area of $O(n \lg n)$ bits of space.

the empirical statistical model for the BWT. Actually, we also conjecture that the $O(n \lg\lg n / \lg_\sigma n)$ term is the minimum additional cost for obtaining the $O(m \lg \sigma)$-time search bound. Bro Miltersen [Mil05] proved a lower bound of $\Omega(n \lg\lg n / \lg n)$ bits for constant-time *rank* and *select* queries on an explicit bitvector (i.e. $\sigma = 2$). (Other tradeoffs for the lower bounds on size are reported in [Mil05, DLO03, GM03].) While this result does not directly imply a lower bound for text indexing, it remains as strong evidence of the difficulty of improving the lower-order terms in our framework since it is heavily based on *rank* and *select* queries.

As another example, consider Theorem 14, where we develop an hybrid implementation of the CSA, occupying $\epsilon^{-1} n H_h + O(n \lg\lg n / \lg_\sigma^\epsilon n)$ bits ($0 < \epsilon \leq 1/3$), so that searching is very fast and takes $O(m / \lg_\sigma n + occ \times \lg^\omega n \lg^{1-\epsilon} \sigma)$ time ($1 > \omega > 2\epsilon/(1-\epsilon) > 0$). For low-entropy text over an alphabet of size $\sigma = O(1)$, we obtain the first self-index that *simultaneously* exhibits sublinear size $o(n)$ in bits and sublinear search and counting query time $o(m)$; reporting the occurrences takes $o(\lg n)$ time per occurrence.

Also, due to the ambivalent nature of our wavelet tree, we can obtain an implementation of the *LF* mapping for the FM-index as a byproduct of our method. (See Section 2.7.3 for more details.) We obtain an $O(m \lg \sigma)$ search/count time by using the backward search algorithm in [FM05] in $nH_h + O(n \lg\lg n / \lg_\sigma n)$ bits. We also get $O(m)$ time in $nH_h + O(n) = nH_h + o(n \lg \sigma)$ bits when σ is not a constant. This avenue has been explored in [FMMN04], showing how to get $O(m)$ time in $nH_h + O(n \lg\lg n / \lg_\sigma n)$ bits when $\sigma = O(\text{polylg}(n))$, using a wavelet tree with a fanout of $O(\lg^\eta n)$ for some constant $0 < \eta < 1$. All these results together imply that the FM-index can be implemented with $O(m)$ search time using nearly optimal space, $nH_h + O(n \lg\lg n / \lg_\sigma n)$ bits, when either $\sigma = O(\text{polylg}(n))$ or $\sigma = \Omega(2^{O(\lg n / \lg\lg n)})$. The space is still $nH_h + O(n) = nH_h + o(n \lg \sigma)$ for the other values of σ, but we do not know if the lower-order term $O(n)$ can be reduced.

2.1.3 Outline of Chapter

The rest of the chapter is organized as follows. In Section 2.2, we describe the differences between various notions of empirical entropy and propose a new definition based on the finite set model. In Sections 2.3–2.7, we describe our algorithmic framework, showing a tighter analysis of the BWT and detailing our new wavelet tree. In Section 2.8, we use this framework to achieve high-order entropy compression in the CSA. In Section 2.9, we apply our CSA to build self-indexing data structures that support fast searching. In Section 2.10, we give some final considerations and open problems.

2.2 High-Order Empirical Entropy

In this section, we formulate our analysis of the space complexity in terms of the high-order empirical entropy of a text T of n symbols drawn from alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$. For ease of exposition, we "number" the symbols in alphabet Σ from 1 to $\sigma = |\Sigma|$, such that the renumbered symbol y is also the yth lexicographically ordered symbol in $\Sigma = \{1, 2, \ldots, \sigma\}$. 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 particular, we discuss various notions of entropy from both an empirical probability model and a finite set model. In Section 2.2.1, we consider classic notions of entropy according to the empirical probability model. We describe a new definition based on the finite set model in Section 2.2.2.

2.2.1 Empirical Probabilistic High-Order Entropy

We provide the necessary terminology for the analysis and explore empirical probability models. For each symbol $y \in \Sigma$, let n^y be the number of its occurrences in text T. With symbol y, we associate its empirical probability, $\mathrm{Prob}[y] = n^y/n$, of occurring in T. (Note that by definition, $n = \sum_{y \in \Sigma} n^y$, so the empirical probability is well defined.) Following Shannon's definition of entropy [Sha48], the 0*th-order empirical entropy* is

$$H_0 = H_0(T) = \sum_{y \in \Sigma} - \mathrm{Prob}[y] \times \lg \mathrm{Prob}[y]. \tag{2.2}$$

Since $nH_0 \leq n \lg \sigma$, expression (2.2) simply states that an efficient variable-length coding of text T would encode each symbol y based upon its frequency in T rather than simply using $\lg \sigma$ bits. The number of bits assigned for encoding an occurrence of y would be $-\lg \mathrm{Prob}[y] = \lg(n/n^y)$.

We can generalize the definition to higher-order empirical entropy, so as to capture the dependence of symbols upon their context, made up of the h previous symbols in the text. For a given h, we consider all possible h-symbol sequences x that appear in the text. (They are a subset of Σ^h, the set of all possible h-symbol sequences over the alphabet Σ.) We denote the number of occurrences in the text of a particular context x by n^x, with $n = \sum_{x \in \Sigma^h} n^x$ as before, and we let $n^{x,y}$ denote the number of occurrences in the text of the concatenated sequence yx (meaning that y precedes x).[2] Then, the *hth-order empirical entropy* is defined as

$$H_h = H_h(T) = \sum_{x \in \Sigma^h} \sum_{y \in \Sigma} - \mathrm{Prob}[y, x] \times \lg \mathrm{Prob}[y|x], \tag{2.3}$$

where $\mathrm{Prob}[y, x] = n^{x,y}/n$ represents the empirical *joint* probability that the symbol y occurs in the text immediately before the context x of h symbols and $\mathrm{Prob}[y|x] = n^{x,y}/n^x$ represents the empirical *conditional* probability that the symbol y occurs immediately before context x, given that x occurs in the text. (We refer the interested

[2]The standard definition of conditional probability for text documents considers the symbol y immediately *after* the sequence x. It makes no meaningful difference, since we could simply use this definition on the reversed text as discussed in [FGMS05].

reader to [CT91] for more details on conditional entropy.) Setting $h = 0$, we obtain H_0 as defined previously. In words, expression (2.3) is similar to (2.2), except that we partition the probability space further according to contexts of length h in order to capture statistically significant patterns from the text.

An important observation to note is that $H_{h+1} \leq H_h \leq \lg \sigma$ for any integer $h \geq 0$. Hence, expression (2.3) states that a better variable-length coding of text T would encode each symbol y based upon the joint and conditional empirical frequency for any context x of y.

Manzini [Man01] gives an equivalent definition of (2.3) in terms of H_0. For any given context x, let w_x be the concatenation of the symbols y that appear in the text immediately before context x. We denote its length by $|w_x|$ and its 0th-order empirical entropy by $H_0(w_x)$, thus defining H_h as

$$H_h = \frac{1}{n} \sum_{x \in \Sigma^h} |w_x| H_0(w_x). \tag{2.4}$$

One potential difficulty with the definition of H_h is that the inner terms of the summation in (2.4) could equal 0 (or an arbitrarily small constant), which can be misleading when considering the encoding length of a text T. (One relatively trivial case is when the text contains n equal symbols, as no symbol needs to be "predicted".) Manzini introduced *modified* high-order empirical entropy H_h^* to address this point and capture the constraint that the encoding of the text must contain at least $\lg n$ bits for coding its length n. Using a modified

$$H_0^* = H_0^*(T) = \max\{H_0, (1 + \lfloor \lg n \rfloor)/n\} \tag{2.5}$$

to make the change, he writes

$$\hat{H}_h = \frac{1}{n} \sum_{x \in \Sigma^h} |w_x| H_0^*(w_x). \tag{2.6}$$

Unfortunately, $\hat{H}_{h+1} \leq \hat{H}_h$ does not necessarily hold in (2.6) as it did for H_h. To solve this problem, let P_h be a *prefix cover*, namely, a set of substrings having length at most h such that every string from Σ^h has a unique prefix in P_h. Manzini then defines the *modified hth-order empirical entropy* as

$$H_h^* = H_h^*(T) = \frac{1}{n} \min_{P_h} \sum_{x \in P_h} |w_x| H_0^*(w_x). \tag{2.7}$$

so that $H_{h+1}^* \leq H_h^*$ *does* hold in (2.7). Other immediate consequences of this encoding-motivated entropy measure are that $H_h^* \geq H_h$ and $nH_h^* \geq \lg n$, but nH_h can be a small constant. Let the *optimal prefix cover* P_h^* be the prefix cover that minimizes H_h^* in (2.7). Thus, Equation (2.7) can be equivalently stated by the expression $H_h^* = \frac{1}{n} \sum_{x \in P_h^*} |w_x| H_0^*(w_x)$.[3]

The empirical probabilities used in the definition of the high-order empirical entropy can be obtained from the number of occurrences $n^{x,y}$, where $\sum_{x \in P_h^*, y \in \Sigma} n^{x,y} = n$. Indeed, $n^y = \sum_{x \in P_h^*} n^{x,y}$ and $n^x = \sum_{y \in \Sigma} n^{x,y}$. This discussion motivates the following definition, which will guide us through our high-order entropy analysis.

Definition 1. The *empirical statistical model* for a text T drawn from an alphabet Σ for contexts of length up to h is composed of two parts stored using $M(T, \Sigma, h)$ bits:

i. The partition of Σ^h induced by the contexts of the prefix cover P_h^*.

ii. The sequence of non-negative integers, $n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}$, where $x \in P_h^*$. (Recall that $n^{x,y}$ is the number of occurrences of yx as a substring of T.)

We denote the number of bits used to store the information in parts (i)–(ii) by $M(T, \Sigma, h)$, as n increases.

2.2.2 Finite Set High-Order Entropy

We provide a new definition of high-order empirical entropy H'_h, based on the finite set model rather than on conditional probabilities. We use this definition to avoid

[3] A minor technical note: h now refers to the length of the longest substring in P_h^*, since no larger value of h can yield a more succinct entropy measure.

dealing with empirical probabilities explicitly. We show that our new definition is $H_h - O(|P_h^*| \lg n) \leq H'_h \leq H_h \leq H_h^*$, so that we can provide bounds in terms of H'_h in our analysis.

For ease of exposition, we "number" the lexicographically ordered contexts x as $1 \leq x \leq \sigma^h$. Let the *multinomial* coefficient $\binom{n}{m_1, m_2, \ldots, m_p} = \frac{n!}{m_1! \, m_2! \cdots m_p!}$ represent the number of partitions of n items into p subsets of size $m_1, m_2, \ldots, m_p$. In this chapter, we define $0! = 1$. (Note that $n = m_1 + m_2 + \cdots + m_p$.) When $m_1 = t$ and $m_2 = n - t$, we get precisely the binomial coefficient $\binom{n}{t}$. We define

$$H'_0 = H'_0(T) = \frac{1}{n} \lg \binom{n}{n^1, n^2, \ldots, n^\sigma}, \tag{2.8}$$

which counts the number of possible partitions of n items into σ unique buckets, i.e. the alphabet size. We use the optimal prefix cover P_h^* in (2.7) to define our *alternative high-order empirical entropy*[4]

$$H'_h = H'_h(T) = \frac{1}{n} \sum_{x \in P_h^*} \lg \binom{n^x}{n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}}. \tag{2.9}$$

For example, consider the text $T = \texttt{mississippi\#}$. Fixing $h = 1$ and taking $P_h^* = \Sigma^h$, we have that all contexts are of length 1. For context $x = \texttt{i}$ occurring $n^{\texttt{i}} = 4$ times in T, we have the symbols $y = \texttt{m}, \texttt{p}$, and $\texttt{s}$ appearing $n^{\texttt{i},\texttt{m}} = n^{\texttt{i},\texttt{p}} = 1$ and $n^{\texttt{i},\texttt{s}} = 2$ times in T. Thus, the contribution of context $x = \texttt{i}$ to $nH'_1(T)$ is $\lg \binom{4}{1,1,2} = \lg 12$ bits. In the next theorem, we show that our formulation of finite set entropy is smaller than the usual definition of empirical probabilistic entropy.

Theorem 1. *For any given text T and context length $h \geq 0$, we have $H'_h \leq H_h$.*

Proof. It suffices to show that $nH'_0 \leq nH_0$ for all alphabets Σ, since we know that $\lg \binom{n^x}{n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}} \leq |w_x| H_0(w_x)$. Setting $P_h^* = \Sigma^h$ in (2.9) and applying Manzini's definition of entropy in (2.4) naturally leads to the claim.

[4]Actually, it can be defined for any prefix cover P_h, including $P_h = \Sigma^h$.

The bound $nH'_0 \leq nH_0$ trivially holds when $\sigma = 1$. We first prove this bound for an alphabet Σ of $\sigma = 2$ symbols. Let t and $n-t$ denote the number of occurrences of the two symbols in T. We want to show that $nH'_0 = \lg \binom{n}{t} \leq nH_0 = t\lg(n/t) + (n-t)\lg(n/(n-t))$ by (2.8). The claim is true by inspection when $n \leq 4$ or $t = 0, 1, n-1$. Let $n > 4$ and $2 \leq t \leq n-2$. We apply Stirling's double inequality [Fel68] to obtain

$$\frac{n^n\sqrt{2\pi n}}{e^{n-1/(12n+1)}} < n! < \frac{n^n\sqrt{2\pi n}}{e^{n-1/12n}}. \tag{2.10}$$

Taking logarithms and focusing on the right-hand side of (2.10), we see that

$$\lg n! < n\lg\frac{n}{e} + \frac{1}{2}\lg n + \frac{1}{12n}\lg e + \lg\sqrt{2\pi}. \tag{2.11}$$

Similarly to (2.11), we take the left-hand side of (2.10), and obtain

$$\lg n! > n\lg\frac{n}{e} + \frac{1}{2}\lg n + \frac{1}{12n+1}\lg e + \lg\sqrt{2\pi}. \tag{2.12}$$

Applying (2.11) and (2.12) to $\lg\binom{n}{t} = \lg(n!) - \lg(t!) - \lg((n-t)!)$, we have

$$nH'_0 = \lg\binom{n}{t} < nH_0 - \frac{1}{2}\lg\frac{t(n-t)}{n} - \lg e\left[\frac{1}{12t+1} + \frac{1}{12(n-t)+1} - \frac{1}{12n}\right] - \lg\sqrt{2\pi}. \tag{2.13}$$

Since $t(n-t) \geq n$ and $1/(12t+1) + 1/(12(n-t)+1) \geq 1/(12n)$ by our assumptions on n and t, it follows that $nH'_0 \leq nH_0$, proving the result when $\sigma = 2$.

Next, we show the claimed bound for the general alphabet ($\sigma \geq 2$ and $h = 0$) and by using induction on the alphabet size (with the base case $\sigma = 2$ as detailed before). We write

$$\lg\binom{n}{n^1, n^2, \ldots, n^\sigma} = \lg\left[\binom{n-n^\sigma}{n^1, n^2, \ldots, n^{\sigma-1}} \times \binom{n}{n^\sigma}\right]. \tag{2.14}$$

We use induction for the right-hand side of (2.14) to get

$$\lg\binom{n-n^\sigma}{n^1, n^2, \ldots, n^{\sigma-1}} \leq \sum_{y=1}^{\sigma-1} n^y \lg\frac{n-n^\sigma}{n^y}, \tag{2.15}$$

$$\lg \binom{n}{n^\sigma} \le n^\sigma \lg \frac{n}{n^\sigma} + (n - n^\sigma) \lg \frac{n}{n - n^\sigma}. \tag{2.16}$$

Summing (2.15) and (2.16), we obtain $\sum_{y=1}^{\sigma} n^y \lg \frac{n}{n^y} = nH_0$, thus proving the claim for any alphabet size σ.

□

The above discussion now justifies the use of H'_h in our later analysis, but we continue to state bounds in terms of H_h as it represents more standard notation. The key point to understand is that we can derive equations in terms of multinomial coefficients without worrying about the empirical probability of symbols appearing in the text T.

2.3 The Unified Algorithmic Framework: Tighter Analysis for the BWT

The characterization of the high-order empirical entropy in terms of the multinomial coefficients given in Section 2.2.2 drives our analysis in a unified framework for text compression and compressed text indexing. In this section, we begin with a simple, yet nearly optimal analysis of the Burrows-Wheeler transform (BWT). Section 2.3.1 formally defines the BWT and highlights its connection to (compressed) suffix arrays. Our key partitioning scheme is described in Section 2.3.2; it serves as the critical foundation in achieving a high-order entropy analysis for the BWT. Sections 2.4.2–2.4.3 motivate and develop our multi-use *wavelet tree* data structure, which serves as a flexible tool in both compression and text indexing. We finish the upper bound analysis of the BWT in Section 2.5, and the lower bound in Section 2.6.

2.3.1 The BWT and (Compressed) Suffix Arrays

We now give a short description of the BWT in order to explain its salient features. Consider the text T = `mississippi#` in the example shown in Table 2.3, where `i` $<$ `m` $<$ `p` $<$ `s` $<$ `#` and `#` is an end-of-text symbol. The BWT forms a conceptual matrix Q whose rows are the cyclic (forward) shifts of the text in sorted order and stores the last column L = `ssmp#pissiii` written as a contiguous string. Moreover, the last column L is an invertible permutation of the symbols in T. In particular, $LF(i) = j$ in Table 2.3 indicates for any symbol $L[i]$, the corresponding position j in F where $L[i]$ appears. For instance, $LF(3) = 5$ since $L[3] =$ `m` occurs in position 5 of F; $LF(8) = 10$ since $L[8] =$ `s` occurs in position 10 of F (as the third `s` among the four appearing consecutively in F).

Using L and LF, we can recreate the text T in reverse order by starting at the last position n (corresponding to `#mississippi`), writing its value from F, and following the LF function to the next value of F. Continuing the example from before, we follow the pointers from $LF(n)$: $LF(12) = 4$, $F[4] =$ `i`; $LF(4) = 6$, $F[6] =$ `p`; $LF(6) = 7$, $F[7] =$ `p`; and so on. In other words, the LF function gives the position in F of the preceding symbol from the original text T. Thus one could store L and recreate T, since we can obtain F by sorting L and the LF function can be derived by inspection. Note that L is compressible using 0th-order compressors, boosting them to attain high-order entropy [FGMS05]. In the following, we connect the BWT with L.

Clearly, the BWT is related to suffix sorting, since the comparison of any two circular shifts must stop when the end marker `#` is encountered. The corresponding suffix array is a simple way to store the sorted suffixes. The suffix array SA for a text T maintains the permuted order of $1, 2, \ldots, n$ that corresponds to the locations of the suffixes of the text in lexicographically sorted order, $T\big[SA[1], n\big]$, $T\big[SA[2], n\big]$, $\ldots$,

$T\big[SA[n], n\big]$. By dropping the symbols *after* `#` in the sorted matrix Q (column '*Sorted*' in Table 2.3), we obtain the sequence of sorted suffixes represented by SA (column '*Suffix Array*' in Table 2.3). In the example above, $SA[6] = 10$ because the sixth largest lexicographically ordered suffix, `pi#`, begins at position 10 in the original text.

We make the connection between the BWT and SA more concrete by describing the neighbor function Φ, introduced to represent the CSA in [GV05]. In particular, the Φ function indicates, for any position i in SA, the corresponding position j in SA such that $SA[j] = SA[i]+1$ (a sort of suffix link similar to that of suffix trees [McC76]). For example in Table 2.3, $\Phi(6) = 4$ since $SA[6] = 10$ and $SA[4] = 11$. As can be seen from Table 2.3, $LF(\Phi(i)) = \Phi(LF(i)) = i$ for $1 \le i \le n$; thus, these functions are *inverses* of each other. Hence, the Φ function is also an invertible representation of the BWT. (The Φ function can also be thought of as the FL mapping while the LF mapping can be thought of as the encoding of inverse suffix links.) Encoding the Φ function is no harder than encoding LF. In the following, we make use of this connection to achieve a high-order empirical entropy analysis of the BWT.

The Φ function can be implemented by using Σ *lists* as shown in [GV05]. Given a symbol $y \in \Sigma$, the list y is the set of positions from the suffix array such that for any position p in list y, $T[SA[p]]$ is *preceded* by y.[5] In words, it collects the positions where y occurs in the text based upon information from the suffix array. The fundamental property of these Σ lists is that each list is an *increasing* series of positions. For instance, list `i` from our example is $\langle 7, 10, 11, 12\rangle$ since for each entry, $T[SA[p]]$ is preceded by an `i`. The concatenation of the lists y for $y = 1, 2, \ldots, \sigma$ gives Φ. Going on in the example, list `m` is $\langle 3\rangle$; list `p` is $\langle 4, 6\rangle$; list `s` is $\langle 1, 2, 8, 9\rangle$, and list `#` is $\langle 5\rangle$. Their concatenation yields the Φ function shown in Table 2.3. Thus, the value of $\Phi(i)$ is just the ith nonempty entry in the concatenation of the lists, and belongs to some list y.

[5]Specifically, $y = T[SA[p]-1]$ for $SA[p] > 1$, and $y = T[n]$ when $SA[p] = 1$.

We can reconstruct SA and the BWT by using Φ and the position f of the last suffix $SA[f] = n$, where $\Phi(f)$ is the position in SA containing the first suffix. Continuing the example from before (where $f = 12$) we can recreate SA by iterating Φ as $\Phi(f) = 5$, $SA[5] = 1$; $\Phi(5) = 3$, $SA[3] = 2$; $\Phi(3) = 11$, $SA[11] = 3$, and so on. In general, we compute $\Phi(f)$, $\Phi(\Phi(f))$, $\ldots$, so that the rank j in SA for the ith suffix in T $(1 \leq i, j \leq n)$ is obtained as $j = \Phi^{(i)}(f)$ by i iterations of Φ on f. However, this process not only recovers the values of SA, but also the corresponding lists y (which provide the symbols for the BWT by the definition of Σ lists). In particular, symbol y occurs in the jth position of the BWT, where $j = \Phi^{(i)}(f)$. In the example, symbol $y = \texttt{#}$ is in position $\Phi(f) = 5$ of the BWT because the fth entry in Φ is in list `#`; symbol $y = \texttt{m}$ is in position $\Phi(5) = 3$ because the fifth entry is in list `m`; symbol $y = \texttt{i}$ is in position $\Phi(3) = 11$, and so on.

2.3.2 Context-Based Partitioning of the BWT

We now show our major result for this section; we describe a nearly optimal analysis of the compressibility of the Burrows-Wheeler transform with respect to high-order empirical entropy, exploiting the relationship between the BWT and suffix arrays illustrated in Section 2.3.1.

Let P_h^* be the optimal prefix cover as defined in Section 2.2, and let $n^{x,y}$ be the corresponding values in Equation (2.9), where $x \in P_h^*$ and $y \in \Sigma$ (see also Definition 1). We denote by $|P_h^*| \leq \sigma^h$ the number of contexts in P_h^*. The following theorem formalizes the bounds that we anticipated in Formulas (1.1) and (2.1) for our analysis.

Theorem 2 (Space-Optimal Burrows-Wheeler Transform). *The Burrows-Wheeler transform for a text T of n symbols drawn from an alphabet Σ can be com-*

pressed using

$$nH_h + M(T, \Sigma, h) \tag{1.1}$$

bits for the best choice of context length h and prefix cover P_h^*, where the number of bits required for encoding the empirical statistical model behind P_h^* (see Definition 1) is

$$M(T, \Sigma, h) \leq \min\left\{g'_h \lg(1 + n/g'_h),\ H_h^* n + \lg n + g''_h\right\}, \tag{2.1}$$

where $g'_h = O(\sigma^{h+1})$ and $g''_h = O(\sigma^{h+1} \lg \sigma^{h+1})$ do not depend on the text length n.

We devote the rest of Section 2.3 and 2.5 to the proof of Theorem 2. We describe our analysis for an *arbitrary* prefix cover P_h, so it also holds also for the optimal prefix cover P_h^* as in Equation (2.9). Since every string in Σ^h has a unique prefix in P_h, it follows that P_h induces a partition of the suffixes stored in the suffix array SA (or the corresponding circular shifts of T). In particular, the suffixes starting with a given context $x \in P_h$ occupy contiguous positions in SA. In the example of Table 2.3, the positions $1, \ldots, 4$ in SA corresponds to the suffixes starting with context $x = \texttt{i}$.

Our basic idea is to apply context partitioning to the Σ *lists* discussed in Section 2.3.1. We implement our idea by partitioning each list y further into sublists $\langle x, y\rangle$ by contexts $x \in P_h$. Intuitively, sublist $\langle x, y\rangle$ stores the suffixes in SA that start with x and are preceded by y. Thus, each item p in sublist $\langle x, y\rangle$ indicates that $T\big[SA[p] - 1, SA[p] + h\big] = yx$. For context length $h = 1$, if we continue the example in Table 2.3, we break the Σ lists by context (in lexicographical order $\texttt{i}$, $\texttt{m}$, $\texttt{p}$, $\texttt{s}$, and $\texttt{\#}$, and numbered from 1 up to $|P_h|$). The list for $y = \texttt{i}$ is $\langle 7, 10, 11, 12\rangle$, and is broken into sublist $\langle 7\rangle$ for context $x = \texttt{p}$, sublist $\langle 10, 11\rangle$ for context $x = \texttt{s}$, and sublist $\langle 12\rangle$ for $x = \texttt{\#}$. We recall that the fundamental property of Σ lists is that each list is an increasing series of positions. Thus, each sublist $\langle x, y\rangle$ we have created is also *increasing* and contains $n^{x,y}$ entries, where $n^{x,y}$ is defined as in Equation (2.9) and Definition 1.

We build a conceptual 2-dimensional table $\mathcal{T}$ that follows Definition 1; see Table 2.4 for an instance of $\mathcal{T}$ on our running example (for $h = 1$). (Each row x implicitly represents the suffixes in SA that start with context x and the columns y are the symbols "predicted" in each context.) The contexts $x \in P_h$ correspond to the rows and the Σ lists y are stored in the columns x. The columns of $\mathcal{T}$ are partitioned by row according to the contexts. Our table $\mathcal{T}$ has some nice properties if we consider its rows and columns as follows:

- We can implement the Φ function by accessing the sublists in $\mathcal{T}$ in *column major order*, as discussed in Section 2.3.1.
- We have a strong relationship with the high-order empirical entropy in Equation (2.9) and the statistical empirical model of Definition 1, if we encode these sublists in *row major order*.

For any context $x \in P_h$, if we encode the sublists in row x using nearly $\lg \binom{n^x}{n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}}$ bits, we automatically achieve the hth-order empirical entropy when summing over all the contexts as required in Equation (2.9). For example, context $x = \texttt{i}$ should be represented with nearly $\lg \binom{4}{1,1,2}$ bits, since two sublists contain one entry each and one sublist contains two entries. The empirical statistical model should record the partition induced by P_h and which sublists are empty, and should encode the lengths of the nine nonempty sublists in Table 2.4, using $M(T, \Sigma, h)$ bits.

The crucial observation to make is that all entries in the row corresponding to a given context x create a *contiguous* sequence of positions. For instance, along the first row of Table 2.4 for $x = \texttt{i}$, there are four entries that are in the range $1 \ldots 4$. Similarly, row $x = \texttt{s}$ contains the four entries in the range $8 \ldots 11$; row $\texttt{s}$ should be encoded with $\lg \binom{4}{2,2}$ bits. We represent this range as an interval $[1, 4]$ with the offset $\#x = 7$. We call this representation a *normalization*, which subtracts the value of $\#x$ from each entry p of the sublists $\langle x, y \rangle$ for $y \in \Sigma$. In words, we normalize the sublists in Table 2.4 by *renumbering each element based on its order within its context* and obtain the context information shown in Table 2.5. Here, n^x is the number of elements in each context x, and $\#x$ represents the partial sum of all prior entries; that is, $\#x = \sum_{x' < x} n^{x'}$. (Note that the values of n^x and $\#x$ are easily computed from the set of sublist lengths $n^{x,y}$.) For example, the first entry in sublist $\langle \mathtt{s}, \mathtt{i} \rangle$, 10, is written as 3 in Table 2.5, since it is the third element in context $\mathtt{s}$. We can recreate entry 10 from $\#x$ by adding $\#\mathtt{s} = 7$ to 3. As a result, each sublist $\langle x, y \rangle$ is a subset of the range implicitly represented by interval $[1, n^x]$ with the offset $\#x$. We exploit this organization to encode the BWT.

Encoding: We run the boosting algorithm from [FGMS05] on the BWT to find the optimal value of context order h and the optimal prefix cover P_h^* using the cost of $nH'_h + M(T, \Sigma, h)$ according to Equation (2.9). (Recall that $H'_h \leq H_h$ by Theorem 1.) Once we know h and set $P_h = P_h^*$, we can cleanly separate the contexts and encode the Φ function as described in our table $\mathcal{T}$. Thus, we follow the two steps below, storing the following components of $\mathcal{T}$:

1. We encode the empirical statistical model given in Definition 1.
2. For each context $x \in P_h$, we separately encode the sublists $\langle x, y \rangle$ for $y \in \Sigma$ to capture high-order entropy. Each of these sublists is a subset of the integers in the range $[1, n^x]$ with offset $\#x$. These sublists form a *partition* of the integers in the interval $[1, n^x]$.

The storage for step 1 is $M(T, \Sigma, h)$, the number of bits required for encoding the model (see Definition 1). The storage required for step 2 should use nearly $\lg \binom{n^x}{n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}}$ bits per context x, and should not exceed a total of nH_h bits plus lower-order terms, once we determine P_h^*, as stated in Theorem 2.

Decoding: We retrieve the empirical statistical model encoded in step 1 above, which allows us to infer the number of rows and columns of our table $\mathcal{T}$, and which sublists are nonempty and their lengths. (Note that the values of n, n^x and $\#x$ can be obtained from these lengths.) Next, we retrieve the sublists encoded in step 2 since we know their lengths. At this point, we have recovered the content of $\mathcal{T}$, allowing us to implement the Φ function with the columns of $\mathcal{T}$ as discussed before. Given Φ, we can decode BWT as described at the end of Section 2.3.1.

We will complete the proof of Theorem 2 in Sections 2.4 and 2.5.

2.4 Encoding Sublists in High-Order Entropy

At the end of Section 2.3.2, we built a partitioning scheme that considers each context $x \in P_h$ independently. In this section, we focus on the problem of encoding the sublists $\langle x, y \rangle$ for $y \in \Sigma$ (i.e., step 2 of encoding). As a reminder, these sublists form a partition of the integers in the range $[1, n^x]$ with offset $\#x$. Moreover, since $\#x$ can be easily inferred using the information from the empirical statistical model in Definition 1, we can recreate the original positions stored in the sublists as usual.

We will encode sublists one context at a time. In other words, we encode the sublists $\langle x, 1 \rangle, \langle x, 2 \rangle, \ldots, \langle x, \sigma \rangle$ at once. One way to do this encodes each context x by encoding the string w_x (from Section 2.2.1), which consists of the symbols y that precede x, concatenated together in BWT order. To encode w_x, we can use is a quasi-arithmetic coder from [HV94] (Theorem 1), requiring $\lg \binom{n^x}{n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}} + 2$ bits of space.

Lemma 1 (Quasi-Arithmetic Coder [HV94]). *Suppose we know the values of n^x and $n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}$ for each context x. We can encode all contexts using one quasi-arithmetic coder for each context x taking just $nH_h + O(\sigma^h)$ bits of space. Decoding any context requires $O(n^x)$ operations on integers of size $O(\sigma)$.*

In the rest of this section, we detail an alternative method of encoding each context x, motivated by applications to text indexing. We begin by encoding each sublist independently, and then evaluate the redundancy of such methods. In particular, Section 2.4.3 describes an important data structure to text indexing, the *wavelet tree*.

2.4.1 Individually Encoded Sublists

In this section, we consider individually encoding each sublist $\langle x, y \rangle$ in context x. Since the positions in each sublist are always increasing, we can represent a sublist $\langle x, y \rangle$ as a subset S of t items drawn from a universe of size n'. In terms of our notation for sublist $\langle x, y \rangle$, $t = n^{x,y}$ and $n' = n^x$. It will also be useful to view the subset S as an *implicit bitvector* B of length n': If S contains the elements $1 \leq s_1 < s_2 < \cdots < s_t \leq n'$, the s_ith entry in the bitvector is $\mathbf{1}$, for $1 \leq i \leq t$ and the remaining $n' - t$ bits are $\mathbf{0}$.

In this section, we will describe two methods: the first uses *t-subset encoding* from [Knu05, Rus05]; the second uses a quasi-arithmetic coder on the bitvector B. We will use t-subsets and this quasi-arithmetic coding scheme heavily over the next few sections; these methods will later be improved in Section 2.4.4.

Encoding Sublists Using *t*-subset Encoding

One method to encode S is to use *t-subset encoding* from [Knu05, Rus05], which requires the information-theoretic minimum of $\lceil \lg \binom{n'}{t} \rceil$ bits. Each t-subset can be encoded or decoded with $O(n')$ operations on large integers. By "large", we mean integers of size $\omega(\lg n)$ bits. All the t-subsets are enumerated in some canonical order (say, lexicographic order) and the rth subset in this order is encoded by the value r written in binary, which requires $\lceil \lg \binom{n'}{t} \rceil$ bits. We will use the following canonical ordering for our subsets: the largest index value r refers to the subset S where the first t positions in the implicit bitvector B are all $\mathbf{1}$.

We now describe an algorithm to take a value r and generate the subset S of t items out of a universe of size n' that r represents. We call this procedure *unranking* the value r. Our algorithm will generate the implicit bitvector B of length n'; each bit position $B[i]$ is initialized to $\mathbf{0}$.

```
function $unrank(B, r, n, t)$ {
  if $(t = 0)$ return B;
  for $(i = 1$ to $n)$
    if $(r > \binom{n-i}{t})$
      $B[i] \leftarrow \mathbf{1}$;
      $r \leftarrow r - \binom{n-i}{t}$;
    $t \leftarrow t - 1$;
}
```

The *unrank* function operates in $O(n')$ time, but uses operations on large integers of size $\omega(\lg n)$ bits. To perform expanded operations, we simply compute B and use brute-force methods to answer queries. The "ranking" algorithm that reverses this process is straightforward.

We highlight the functions *rank* and *select* as two advanced operations of particular interest, since they are often used in our remaining data structures. For a bitvector B of size n', the function $rank_{\mathbf{1}}(B, i)$ returns the number of $\mathbf{1}$s in B up to (and including) position i. The function $select_{\mathbf{1}}(B, i)$ returns the position of the ith $\mathbf{1}$ in B. We can also define $rank_{\mathbf{0}}$ and $select_{\mathbf{0}}$ in terms of the $\mathbf{0}$s in B.

When we have t-subsets, we can support *rank* and *select* by unranking the index value r into its implicit bitvector B, and then performing a brute-force linear walk to return the correct answer.[6] We summarize these results into the following lemma.

[6]Time bounds are not the issue at this stage; we address these concerns in Sections 2.4.4 and 2.7.1.

Lemma 2 (t-subset Encoding). *Let the subset S consist of t items drawn from a universe of size n, where we already know t and n (and do not need to encode them). Then, we can use t-subset encoding to represent S using $\lg \binom{n}{t} + O(1)$ bits of space and can be encoded or decoded using $O(n)$ operations on large integers, i.e., integers of size $\omega(\lg n)$ bits.*

Encoding Sublists Using Quasi-Arithmetic Encoding

As we saw in Section 2.4.1, using t-subset encoding to represent a subset S with t items drawn from a universe of size n' requires $O(n)$ operations on large integers. To avoid the large integer computations, we can use a quasi-arithmetic coder [HV94] to encode or decode the implicit bitvector B. The coder will sequentially encode the positions of B, encoding whether each bit is a $\mathbf{0}$ or a $\mathbf{1}$. At any step of the encoder, the probability of the next bit being a $\mathbf{1}$ is t/n' (for the current values of t and n'); the probability of the next bit being a $\mathbf{0}$ is $1 - t/n'$. We summarize this scheme in the following lemma.

Lemma 3 (Quasi-Arithmetic Subset Encoding). *Let the subset S consist of t items drawn from a universe of size n, where we already know t and n (and do not need to encode them). Let B be the implicit bitvector related to S. Then, we can use a quasi-arithmetic coder to represent S (by encoding B) using $\lg \binom{n}{t} + O(1)$ bits of space and can be encoded or decoded using $O(n)$ operations on small integers.*

To support *rank* and *select*, we will use a brute-force method once we have recovered B. We reduce the encoding and decoding time for storing a subset S from $O(n')$ time to $O(t)$ time in Section 2.4.4, where t represents the number of items in subset S (or equivalently, the number of $\mathbf{1}$s in B).

2.4.2 The Space Redundancy of Encoding Multiple Sublists

In this section, we revisit encoding each sublist $\langle x, y\rangle$ independently of the others. One (simple) method would be to encode each sublist $\langle x, y\rangle$ as a subset of $t = n^{x,y}$ items out of a universe of $n' = n^x$ items using t-subset encoding or quasi-arithmetic coding, described in Section 2.4.1. To encode and decode sublist $\langle x, y\rangle$, we can use subset rank and unrank primitives (respectively) on a sequence r of $\lceil \lg \binom{n^x}{n^{x,y}} \rceil$ bits, or encode or decode the implicit bitvector B related to sublist $\langle x, y\rangle$.

Unfortunately, despite the fact that t-subset encoding (and quasi-arithmetic coding) is *locally* optimal for sublist $\langle x, y\rangle$ does not imply that it is *globally* optimal for encoding all the sublists together. In fact, summing the size of subset encodings for all the sublists shows that the total space adds an $O(n)$ term to the entropy bound nH_h! This finding is given in the following lemma. First, we briefly define some useful notation. Let t_x be the number of nonempty sublists contained in a given context x and, without loss of generality, let the number of entries in the nonempty sublists be $n^{x,1}$, $n^{x,2}$, …, n^{x,t^x}, where $\sum_{1\le y\le t^x} n^{x,y} = n^x$.

Lemma 4. *Given context x, the following relation holds,*

$$\sum_{1\le y\le t^x} \lg \binom{n^x}{n^{x,y}} = \lg \binom{n^x}{n^{x,1}, n^{x,2}, \ldots, n^{x,t^x}} + O(n^x). \tag{2.17}$$

Proof. When $t^x = 2$, $\sum_{1\le y\le t^x} \lg \binom{n^x}{n^{x,y}} = \lg \binom{n^x}{n^{x,1}, n^{x,2}, \ldots, n^{x,t^x}}$ and the lemma is trivially proved. Thus, let $t^x > 2$, so that the following holds.

$$\begin{aligned}
\sum_{1\le y\le t^x} \lg \binom{n^x}{n^{x,y}} &= \lg \left[\left(\frac{1}{n^{x,1}!\, n^{x,2}! \ldots n^{x,t^x}!} \right) \frac{(n^x!)^{t^x}}{(n^x - n^{x,1})!\, (n^x - n^{x,2})! \ldots (n^x - n^{x,t^x})!} \right] \\
&\le \lg \left[\frac{1}{n^{x,1}!\, n^{x,2}! \ldots n^{x,t^x}!} (n^x)^{(n^x)} \right] \\
&= \lg \left[\frac{1}{n^{x,1}!\, n^{x,2}! \ldots n^{x,t^x}!} \right] + n^x \lg n^x
\end{aligned}$$

Since $\lg \binom{n^x}{n^{x,1}, n^{x,2}, \ldots, n^{x,t^x}} = \lg \left[\frac{n^x!}{n^{x,1}!\, n^{x,2}! \ldots n^{x,t^x}!} \right]$ and $\lg n^x! \le n^x \lg n^x - n^x \lg e + 1/2 \lg n^x + 1/12n \lg e + \lg \sqrt{2\pi}$ by Stirling's inequality [Fel68], the claim is proved.

The additional term of $O(n^x)$ in Equation (2.17) is tight in several cases; for example, when $t^x = n^x > 2$ and each $n^{x,y} = 1$. $\square$

The apparent paradox implied by Equation (2.17) can be resolved by realizing that each subset encoding only represents the entries of one particular sublist; that is, there is a separate subset encoding for each symbol y in context x. In the multinomial coefficient of Equations (2.9) and (2.17), all the sublists are encoded together as a multiset. Thus, it is more expensive to have a subset encoding of *each sublist individually* rather than having a single encoding for the entire context. In Lemma 4, the $O(n^x)$ additional bits account for the extra cost incurred by encoding, for each sublist $\langle x, y\rangle$, not only the positions of w_x where y appears, but also the positions where it does *not* appear. When summed over all n entries in all sublists and all contexts, this term gives an $O(n)$ contribution to the total space bound.

To avoid this excess encoding cost, we perform a *scaling* of the universe. For context x, we apply the scaling of the universe as follows. When we encode sublist $\langle x, y\rangle$, we only encode its positions in terms of positions *not* used by sublists $\langle x, y'\rangle$ for $1 \le y' < y$. (These positions are those corresponding to the remaining **0**s in the resulting bitvector.) In this way, we iterate the scaling to the sublists:

1. We represent sublist $\langle x, 1\rangle$ using $n^{x,1}$-subset encoding in a universe of size n^x, using $\lceil \lg \binom{n^x}{n^{x,1}} \rceil$ bits.
2. For $y = 2, 3, \ldots, t^x$, we represent sublist $\langle x, y\rangle$ using $n^{x,y}$-subset encoding in a scaled universe of size $n' = n^x - \sum_{y'=1}^{y-1} n^{x,y'}$, with $\lceil \lg \binom{n'}{n^{x,y}} \rceil$ bits.

We give an example using Table 2.5 for context $x = \texttt{i}$. Here, sublist $\langle x, \texttt{m}\rangle$ contains the third position in the interval $[1, 4] = \{1, 2, 3, 4\}$; the corresponding bitvector **0010** is encoded in $\lceil \lg \binom{4}{1} \rceil$ bits. When we encode sublist $\langle x, \texttt{p}\rangle$, we only encode its positions in terms of positions *not* used by sublist $\langle x, \texttt{m}\rangle$. In other words, we are encoding which of the remaining positions $\{1, 2, 4\}$ (corresponding to the **0**s in the bitvector for sublist $\langle x, \mathtt{m}\rangle$) contain the symbol p. In this case, entry 4 in sublist $\langle x, \mathtt{p}\rangle$ sublist corresponds to the third remaining position in $\{1, 2, 4\}$ (out of three items in the scaled universe). The resulting bitvector **001**, is encoded in $\lceil \lg \binom{3}{1} \rceil$ bits. The only remaining positions now are $\{1, 2\}$, corresponding to the two remaining **0**s in the scaled universe. To encode sublist $\langle x, \mathtt{s}\rangle$, we only encode those positions not used by sublists $\langle x, \mathtt{m}\rangle$ and $\langle x, \mathtt{p}\rangle$. Sublist $\langle x, \mathtt{s}\rangle$ contains the remaining available positions and we implicitly encode the bitvector **11** encoded in $\lceil \lg \binom{2}{2} \rceil = 0$ bits of space. The total number of bits for context x is $\lceil \lg \binom{4}{1} \rceil + \lceil \lg \binom{3}{1} \rceil + \lceil \lg \binom{2}{2} \rceil < \lg \binom{4}{1,1,2} + 3$ as required.

To recover the 2nd position in the $\langle \mathtt{i}, \mathtt{s}\rangle$ sublist, we have to find the position j of the 2nd non-position in the the $\langle \mathtt{i}, \mathtt{p}\rangle$ sublist (i.e. the position j of the 2nd **0** in its corresponding bitvector). For this example, we can see that $j = 2$. Then we have to find the position of the 2nd non-position in the $\langle \mathtt{i}, \mathtt{m}\rangle$ sublist, and so on, cascading the query until an answer is reached. Finding the right position in the bitvectors uses a *rank* or *select* query (which we use more when we discuss text indexing).

Note that the last sublist, $\langle x, t^x\rangle$, is encoded using $\lceil \lg \binom{n^{x,t^x}}{n^{x,t^x}} \rceil = 0$ bits. We introduce the notion of *depth* of a context x, which measures the maximum number of sublists in context x that must be examined to recover the entries of any sublist of x. As we shall see later, the depth is related to decompression time; in the above scheme, the depth is t^x. The lemma below captures the time and space required for our incremental representation scheme.

Lemma 5 (Incremental Representation of Sublists). *Using the incremental representation of sublists by scaling the universe, we can encode the t^x nonempty sublists for each context x in fewer than $\lg \binom{n^x}{n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}} + t^x$ bits, so that the depth is t^x.*

Proof. We show that the information theoretically minimum space required to encode all sublists in context x is

$$\left\lceil \lg \binom{n^x}{n^{x,1}} \right\rceil + \left\lceil \lg \binom{n^x - n^{x,1}}{n^{x,2}} \right\rceil + \left\lceil \lg \binom{n^x - n^{x,1} - n^{x,2}}{n^{x,3}} \right\rceil + \cdots + \left\lceil \lg \binom{n^{x,t^x}}{n^{x,t^x}} \right\rceil$$
$$< \lg \left[\binom{n^x}{n^{x,1}} \binom{n^x - n^{x,1}}{n^{x,2}} \binom{n^x - n^{x,1} - n^{x,2}}{n^{x,3}} \cdots \binom{n^{x,t^x}}{n^{x,t^x}} \right] + t^x$$
$$= \lg \left[\frac{n^x!}{n^{x,1}!\, n^{x,2}! \cdots n^{x,t^x}!} \right] + t^x = \lg \binom{n^x}{n^{x,1}, n^{x,2}, \ldots, n^{x,t^x}} + t^x.$$

We can replace t^x by σ in the multinomial coefficient of the above formula because the empty sublists do not contribute. The depth of the above approach is sequential in terms of t_x, the number of nonempty sublists within x. Thus, the depth is t^x, since we potentially have to backtrack through each nonempty sublist to recover the entries of the last sublist in the context. □

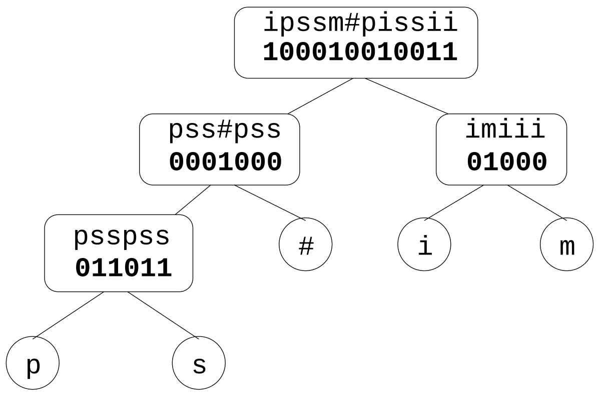


Figure 2.1: An example wavelet tree.

2.4.3 The Wavelet Tree

As we saw in Section 2.4.2, the linear representation of sublists in Lemma 5 for context x may requires up to t^x queries on nonempty sublists to decode an answer. We instead provide the *wavelet tree* data structure, which is of independent interest, that reduces the number of queries on nonempty contexts to just $\lg t^x \leq \lg \sigma$. A wavelet tree is a binary tree structure that reduces the compression of a string from alphabet Σ to the compression of σ binary strings. We now describe the wavelet tree data structure for any string T of length n drawn from an alphabet Σ.

Our wavelet tree data structure is a complete binary tree with σ leaves, one for each symbol appearing in T. For each internal node u of this binary tree, we associate two vectors of the same length: a text vector T_u composed of symbols drawn from Σ, and bitvector B_u. At the root node r, T_r and B_r are both of length n. At the root, we set $T_r = T$. Let Σ_L be the lexicographically smallest $\lceil \sigma/2 \rceil$ symbols present in T, and Σ_R be the lexicographically largest $\lfloor \sigma/2 \rfloor$ symbols present in T. Then, we set $B_r[i] = \mathbf{0}$, if $T_r[i] \in \Sigma_L$, and $B_r[i] = \mathbf{1}$ otherwise.

We recurse this process on the $n_{\mathbf{0}}$ positions containing a symbol in Σ_L for the left subtree of r, and on the $n_{\mathbf{1}}$ positions containing a symbol in Σ_R for the right subtree of r. The text vector for left child r_l is the concatenation of the symbols j such that $B_r[j] = \mathbf{0}$. The right child is processed similarly. The wavelet tree data structure only stores B_u for each node; it stores it in some compressed form, such as t-subset encoding (described in Section 2.4.1).

To explain our wavelet tree data structure more clearly, we will refer to the example in Figure 2.1, built on the BWT of the string `mississippi#`. Here, each internal node u consists of the two vectors T_u and B_u. (We have drawn the leaves here for clarity, though they are not needed in the wavelet tree.) Suppose we wanted to know which symbol appears in text position 9 (which is an `s` in this example).‘ We observe that $B_r[9] = \mathbf{0}$, which tells us that the correct symbol is contained in $\Sigma_L = \{\texttt{p}, \texttt{s}, \texttt{\#}\}$. Furthermore, since the 9th position of B_r is the *sixth* $\mathbf{0}$, we know that our answer corresponds to the 6th position on the left child c_1. (Computing this information requires a *rank* query, which we will describe in detail in Section 2.7.2, when we want fast access. For now, we explicitly compute it using t-subsets or a quasi-arithmetic coder in a brute-force way, as described in Section 2.4.1.) We proceed to search in the

6th position. $B_{c_1}[6] = \mathbf{0}$ which means the correct symbol is contained in $\Sigma_L = \{\mathtt{p}, \mathtt{s}\}$, so we again go to the left subchild c_2, searching for the 5th position there. Here, we find that $B_{c_2}[5] = \mathbf{1}$, which leads us to the leaf representing s, which we return as the answer.

The key observation is to note that each of the $t^x - 1$ internal nodes represents elements relative to its subtrees. Rather than the linear relative encoding of sublists we had in Section 2.4.2, we use a tree structure to reduce the dependency on previously encoded information. In particular, to decode any particular sublist in a wavelet tree, a query would only need to access $O(\lg t^x)$ internal nodes in a balanced wavelet tree. In some sense, the earlier approach corresponds to a completely skewed wavelet tree, as opposed to the balanced structure now. Recovering the entries of any sublist $\langle x, y \rangle$ proceeds exactly as in Section 2.4.2, except that we start from the leaf corresponding to sublist $\langle x, y \rangle$ and examine only the subsets in its ancestors.

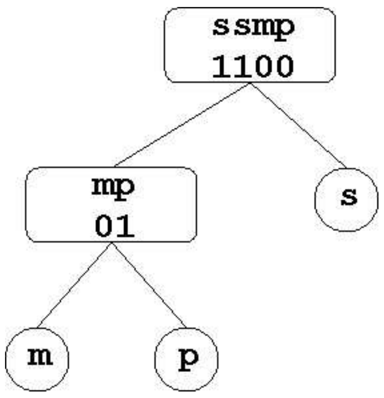


Figure 2.2: A wavelet tree for context i in our example.

Interestingly, any shape of the wavelet tree gives the same upper bounds on space; the only aspect that changes from an altered shape is the number of queries required. We give a short example for context i on our continuing example in Figure 2.2. We group sublists $\langle \mathtt{i}, \mathtt{m} \rangle$ and $\langle \mathtt{i}, \mathtt{p} \rangle$ together, thus obtaining positions $\langle 3, 4 \rangle$ for them. For this grouping, the corresponding bitvector would be **1100**, represented with $\lceil \lg \binom{4}{2} \rceil$

bits. Then, the $\langle \texttt{i}, \texttt{s} \rangle$ list would be represented as before, with $\lceil \lg \binom{2}{2} \rceil$ bits. We need a further subset encoding to distinguish between $\langle \texttt{i}, \texttt{m} \rangle$ and $\langle \texttt{i}, \texttt{p} \rangle$, but on a scaled universe with bitvectors **01** and **1**, respectively, using $\lceil \lg \binom{2}{1} \rceil$ and $\lceil \lg \binom{1}{1} \rceil$ bits. The total space is still bounded as before, namely, $\lceil \lg \binom{4}{2} \rceil + \lceil \lg \binom{2}{2} \rceil + \lceil \lg \binom{2}{1} \rceil + \lceil \lg \binom{1}{1} \rceil < \lg \binom{4}{1,1,2} + 3$, since the terms of the form $\lceil \lg \binom{k}{k} \rceil = 0$ do not contribute. With this intuition firmly in mind, we now detail the general lemma and its proof.

Lemma 6 (Wavelet Tree Compression). *Suppose we know the values of n^x and $n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}$. Using a wavelet tree for each context x, we can encode the t^x nonempty sublists for context x in fewer than $\lg \binom{n^x}{n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}} + t^x$ bits, so that the depth is $O(\lg t^x)$.*

Proof. We analyze the space required in terms of the contribution of each internal node's t-subset encoding. We prove that this cost is the logarithm of the multinomial coefficient in Equation (2.9) for the high-order empirical entropy.[7] Note that the leaves of the wavelet tree do not contribute to the cost since they generate terms of the form $\lceil \lg \binom{n^{x,y}}{n^{x,y}} \rceil = 0$ in the calculations for the number of required bits. By induction, it is simple to verify that the space required among all the $t^x - 1$ internal

[7]In some sense, we are calculating the space requirements for each sublist $\langle x, y \rangle$, propagated over the entire tree. For instance, in the example above, $\langle x, y \rangle$ is implicitly stored in each node of its root-to-leaf path. We could analyze it this way and show that the two notions are the same, though we defer the argument in the interest of brevity.

nodes is

$$\begin{aligned}
&\lg\binom{n^{x,1}+n^{x,2}}{n^{x,2}} + \lg\binom{n^{x,3}+n^{x,4}}{n^{x,4}} + \cdots + \lg\binom{n^{x,t^x-1}+n^{x,t^x}}{n^{x,t^x}} \\
&\quad + \lg\binom{n^{x,1}+\cdots+n^{x,4}}{n^{x,3}+n^{x,4}} + \lg\binom{n^{x,5}+\cdots+n^{x,8}}{n^{x,7}+n^{x,8}} + \cdots + \lg\binom{n^{x,t^x-3}+\cdots+n^{x,t^x}}{n^{x,t^x-1}+n^{x,t^x}} \\
&\quad \vdots \\
&\quad + \lg\binom{n^{x,1}+\cdots+n^{x,t^x}}{n^{x,1}+\cdots+n^{x,t^x/2}} = \lg\binom{n^x}{n^{x,1},n^{x,2},\ldots,n^{x,t^x}} \\
&\quad = \lg\binom{n^x}{n^{x,1},n^{x,2},\ldots,n^{x,\sigma}}.
\end{aligned}$$

Hence, each wavelet tree encodes a particular context in precisely the high-order empirical entropy, which is what we wanted in Equation (2.9). As in the proof of Lemma 5, the rounding due to the ceilings adds further t^x bits to the above bound. □

Lemma 6 is a key result for many applications, such as text indexing and range searching. One of its more subtle contributions is in achieving a near-optimal 0th-order compressor using a series of optimally-stored succinct dictionaries. The connection between these two concepts is a recurring theme in state-of-the-art BWT compression. The wavelet tree serves as a natural way to express the 0th-order entropy of a string with alphabet Σ using several strings with a binary alphabet.

The advantage of using the wavelet tree for text indexing will be clear in the rest of the chapter, where we use the Φ function described in Section 2.3.1. In Section 2.7, we will replace the t-subset encodings with fully indexable dictionaries [RRR02] inside the nodes of the wavelet tree. We will exploit its organization for compressed text indexing, as we detail in Sections 2.8 and 2.9.

One problem with our current implementation of the wavelet tree is its use of subset encoding using t-subsets or quasi-arithmetic coders, requiring $O(n)$ operations. To solve this problem, we introduce our subset encoder in Section 2.4.4, which is a data structure of independent interest. It will replace the subset encodings the wavelet tree, without adding any additional space.

2.4.4 Subset Encoding With Small Integers

In this section, we describe a technique for subset encoding, storing a set S of t items out of a universe of size n such that it can be encoded or decoded using $O(t)$ operations on small integers. This goal is an improvement over Lemma 3, which requires $O(n)$ such operations. We assume that each of the n elements of the universe appears equally likely as an element of the set S. This assumption is not a debilitating one; in fact, the sublists that we store in the previous sections use the same assumption. As we described in Section 2.4, we can think of a t-subset as a succinct way to store an implicit bitvector. We could encode this bitvector using arithmetic (or quasi-arithmetic) coding using Lemma 1, but encoding/decoding would still require $O(n)$ operations.

Another approach is to encode the *gaps* between the items $s_1, s_2, \ldots, s_t$ that appear in the set S. (The ith gap is formally $s_i - s_{i-1}$, where $s_0 = 1$.) To encode the items, we associate a probability distribution for the different gap values, and encode each gap according to its probability using any of a number of techniques (say, for instance, the quasi-arithmetic coder from [HV94]). Using this method, the items are decoded *sequentially* using $O(t)$ operations on small integers of size $O(\lg n)$ bits.

The gaps are encoded sequentially. For this section, we redefine t to be the number of items *left* to encode out of a remaining universe of size n. In other words, the values of n and t will scale as we sequentially encode gaps. (As described in [Vit84], this scaling definition of n and t will not be a problem.) Let X be the random variable that determines the length of the next gap value to be encoded. Note that the range of X is the set of integers in the interval $1 \leq x \leq n - t$. We will restrict gaps to a length at most n/t and aggregate the probabilities of larger gaps into a single escape gap. If we need to encode an escape gap of length $g > n/t$, we reset $n = n - g - 1$ and continue processing. Hence, the range for X is $1 \leq x \leq n/t$.

One approach is to encode the gap X using the exact discrete probabilities $f(x)$. The probability distribution function (pdf) for $f(x)$ is

$$f(x) = \begin{cases} \alpha_1 \frac{t}{n} \frac{(n-x-1)^{\underline{t-1}}}{(n-1)^{\underline{t-1}}} & \text{if } 1 \leq x < n/t; \\ \alpha_2 \frac{(n-n/t-2)^{\underline{t}}}{n^{\underline{t}}} & \text{if } x = n/t, \end{cases}$$

where we use the notation $a^{\underline{b}}$ to denote the *falling power* $a(a-1)\ldots(a-b+1) = a!/(a-b)!$. The constants α_1, α_2 are normalization factors so that $f(x)$ sums up to 1. The probability for $x = n/t$ includes the sum of all probabilities for $x > n/t$. Both the expected value and the standard deviation of X are roughly n/t, which is, as expected, the average gap length. We could use this distribution to encode gaps in the quasi-arithmetic coder [HV94]; however, computing $f(x)$ will require large integer computations.

To avoid these computations, we are willing to use a (continuous) approximation of $f(x)$ to *encode* the gaps, though they still are occur with probability $f(x)$. Using an approximate distribution will incur some additional encoding overhead, which we will analyze later in this section. In particular, we will approximate $f(x)$ with two probability estimates $g_1(x)$ and $g_2(x)$ from [Vit84] that are easy to compute using built-in logarithm functions. In the rest of this section, we will assume the use of b-bit arithmetic, where b is an appropriately large constant multiple of $\lg n$, such that exponential and logarithm functions are correct to b bits. Then, the absolute and relative error of computing a constant number of such functions can be bounded by $O(1/n^c)$ for some constant c. Furthermore, the quasi-arithmetic coder we will use requires at most $O(1/n)$ extra bits of storage per gap stored [HV94]. Thus, we focus on the final source of error, the approximation itself.

Our technique will use a quasi-arithmetic coder that will encode each gap using $g_1(x)$ or $g_2(x)$ instead of $f(x)$. To use these correctly, we need two further properties from each approximation function:

- Given a gap x, we need to determine in $O(1)$ time the endpoints of an interval whose length approximates $f(x)$. To account for this goal, each of our approximation functions will be continuous. Then, for a probability function $g(x)$, we can compute the endpoints $G(x)$ and $G(x-1)$, where $G(x) = \int g(x)$ is the cumulative distribution function.

- Each interval $G(x) - G(x-1)$ must be of length at least $O(1/n^c)$ for some constant c, so that we can represent it using $c \lg n$ bits.

The first property will be obvious for our choices for $g(x)$; we will prove the second property for each case.

Our two approximation functions are $g_1(x)$ and $g_2(x)$. We will use $g_1(x)$ to encode gaps when $t \leq \sqrt{n}$, and $g_2(x)$ to encode gaps when $t > \sqrt{n}$. If $t > n/2$, we reverse the role of **0**s and **1**s (i.e., encode gaps of **1**s), set $t = n - t$, and proceed as above. (Obviously, if $t = 0$, we do not generate any more gaps.) We will use these probability estimates in the quasi-arithmetic coder to encode the gaps, alternating between g_1 and g_2 and maintaining t as necessary to operate within the above constraints. We do not have to remember which estimate was used to encode each item or when we complemented the set S (which could take a lot of space to encode), since it can be easily determined during the encoding or decoding process.

Finally, we must analyze how many extra bits we will take to encode the gap X using $g_1(x)$ or $g_2(x)$ rather than $f(x)$. We define $g_1(x)$ as

$$g_1(x) = \begin{cases} \beta_1 \frac{t}{n} \left(1 - \frac{x}{n}\right)^{t-1} & \text{if } 1 \leq x < n/t; \\ \beta_2 e^{-1} & \text{if } x = n/t, \end{cases}$$

where β_1 and β_2 are normalization factors so that $g_1(x)$ sums up to 1. The random variable X_1 with probability density $g_1(x)$ has the beta distribution, scaled to the universe $[1..n]$ with parameters $a = 1$ and $b = t$. Notice that X_1 can be generated quickly with only one uniform or exponential random variable [Vit84]. We similarly define $g_2(x)$ as

$$g_2(x) = \begin{cases} \beta_3 \left(1 - \frac{t-1}{n-1}\right)^x \ln\left(1 - \frac{t-1}{n-1}\right) & \text{if } 1 \leq x < n/t; \\ \beta_4 e^{-1} & \text{if } x = n/t, \end{cases}$$

where β_3 and β_4 are normalization factors so that $g_2(x)$ sums up to 1. The random variable X_2 with probability density $g_2(x)$ has the exponential distribution. Here, X_2 can be generated quickly with only one uniform or exponential random variable [Vit84].

We now show that the probability of any event is at least $\Omega(1/n^c)$, for some constant c. This will allow us to use a quasi-arithmetic coder that makes use of a word size of roughly $c \lg n$ bits.

Lemma 7. *The approximation functions $g_1(x)$ and $g_2(x)$ are at least $\Omega(1/n^c)$ for any given gap x.*

Proof. We show the result for $g_1(x)$ first; it suffices to show the result for the case when $g_1(x)$ is minimized, namely when $x = n/t$. We write

$$g_1(n/t) = \frac{t}{(n - n/t)} \left(1 - \frac{1}{t}\right)^t.$$

As $t \to \infty$, this becomes $(1/(n - n/t))(1/e) = O(1/n^c)$, which can be represented using $c \lg n$ bits of space. For smaller t, $g_1(n/t)$ is strictly larger than the limit, since $(1 - 1/t) < 1$. (In the degenerate case where $t = 1$, we handle the case separately.)

A similar analysis applies for $g_2(x)$ as well. It again suffices to show the result when $g_(x)$ is minimized; this happens when $x = n/t$. We write

$$g_2(n/t) = \frac{(1 - (t-1)/(n-1))^{n/t}}{\ln(1 - (t-1)/(n-1))^{n/t}}.$$

Since $(n-1)/(t-1) \leq n/t$, we can easily see that this case is similar to the previous case, thus proving the lemma. $\square$

Using a $g_1(x)$ and $g_2(x)$ that can be represented using only $c \lg n$ bits, we can encode and decode the set S using $O(t)$ operations on *small* integers. To quickly arrive at the endpoints within the unit interval corresponding to a particular gap x, we can use the cumulative distribution function for these pdfs.

However, we still have the problem that we may spend additional bits to encode each gap, since our probabilities are only estimates for the pdf $f(x)$. We address this point in the following lemma, and show that the worst-case difference between the encodings is quite small.

Lemma 8. *Suppose each of the t items of subset S is drawn from $[1..n]$. The extra bits needed to encode all gaps using the probability estimates g_1 when $t \le \sqrt{n}$ and g_2 when $t > \sqrt{n}$ instead of f, is at most $O(1)$ bits using an arithmetic coder.*

Proof. We look at the worst case where we encode a gap $X = x$ using the probability estimate $g_1(x)$ or $g_2(x)$ rather than $f(x)$. First, we look at the scenario for $g_1(x)$. The extra bits needed to encode any gap using $g_1(x)$ is $|\lg(f(x)) - \lg(G_1(x) - G_1(x-1))|$. Since $g_1(x)$ is a decreasing function, we can upper-bound the second lg-term with $g_1(x-1)$ and lower bound it by $g_1(x)$. We will now show both bounds, thus giving us the result we want. For the upper bound, we write

$$\begin{aligned} \lg(f(x)/g_1(x-1)) &\le \lg\left(\frac{n-x-1}{(n-1)(1-(x-1)/n)}\right)^{t-1} \\ &= \lg\left(1 + \frac{1-n-x}{n^2 - xn + x - 1}\right)^{t-1}. \end{aligned}$$

Let $y = (1-n-x)/(n^2 - xn + x - 1)$. Since $0 \le y < 1$, we can see that the extra bits we require are $\lg(f(x)/g_1(x)) < (t-1)(\lg e)[y - y^2/2]$, since $\ln(1+y) < y - y^2/2$ for all $|y| < 1$. The worst-case ratio of $f(x)/g_1(x)$ occurs when $x = n/t$. Substituting and using simple algebra, we see that we need an additional $O(t/n)$ bits per item, or $O(t^2/n)$ bits for all t items. (For the special case where $t = 1$, we encode it using

$O(1)$ extra bits, but this happens only once.) We use $g_1(x)$ when $t \leq \sqrt{n}$, so this contributes at most $O(1)$ bits for all t gaps.

Now we lower bound the extra bits needed to encode any gap using $g_1(x)$. We write

$$\begin{aligned} \lg(f(x)/g_1(x)) &\geq \lg\left(\frac{n-x-t+1}{(n-t+1)(1-x/n)}\right)^{t-1} \\ &= \lg\left(1+\frac{x(t-1)}{n^2+n-nx-nt+x-tx}\right)^{t-1}. \end{aligned}$$

The worst-case ratio of $f(x)/g_2(x)$ occurs when $x = n/t$. Substituting and using simple algebra, we require at most $O(t/n)$ extra bits per item. We only use g_1 when $t \leq \sqrt{n}$, so this contributes at most $O(1)$ bits for all t gaps.

A similar separation and analysis also applies to the overhead for $g_2(x)$, thus proving the lemma. $\square$

Putting these results together, we arrive at the following theorem, which describes our subset-encoding scheme.

Theorem 3 (Subset Encoding With Small Integers). *Suppose each of the t items of subset S is drawn from the universe $[1..n]$, where we already know t and n (and do not need to encode them). Then, there exists an encoding of subset S that requires $\lg\binom{n}{t} + O(1)$ bits of space and can be encoded or decoded using $O(t)$ operations on small integers.*

2.5 Encoding the Empirical Statistical Model

In this section, we will provide an analysis of the encoding length of the empirical statistical model, thus finishing the proof of Theorem 2. Our scheme was divided into two components: the encoding of a series of small disjoint subtexts (or sublists), one for each context x, and the encoding of the length of each subtext, together with the statistics of each subtext. We did not analyze the cost required to store this empirical statistical information. We briefly recap now:

- For each context x, the storage for step 2 uses fewer than $\lg \binom{n^x}{n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}} + t^x$ bits by Lemma 1 and 6. We use Equation (2.9) and Theorem 1 to bound the above term by $nH'_h + |P^*_h|\sigma$ for all contexts $x \in P^*_h$ in the worst case. Since $|P^*_h| \leq \sigma^h$, we bound the space required to store the BWT by $nH_h + \sigma^{h+1}$ bits.
- To decode the succinct dictionaries in step 2, we need to know the number of symbols of each type stored in each subtext (sublist) for context x. Collectively, this information maintains the empirical statistical model that allow us to achieve hth order entropy with our scheme. We call its encoding length $M(T, \Sigma, h)$ (in bits), and we are interested in discovering just how succinctly this information can be stored. Thus, the storage for step 1 is $M(T, \Sigma, h)$ bits.[8]

Our storage of the BWT requires $nH_h + \sigma^{h+1} + M(T, \Sigma, h)$ bits, and bounding the quantity $M(T, \Sigma, h)$ may help in understanding the compressible nature of the BWT. We will devote the rest of this section to developing two bounds for the storage of empirical statistical model. One benefit of pursuing bounds in this framework is that it simplifies the burden of analysis: namely, it translates the overhead costs of the BWT into the cost for encoding the integer lengths $n^{x,y}$.

2.5.1 Definitions and a Simple Bound

In this section, we describe a simple encoding for the empirical statistical model, which takes $M(T, \Sigma, h)$ bits to encode. Recall from Definition 1 in Section 2.2.1 that the empirical statistical model encodes two items: the partition of Σ^h induced by the optimal prefix cover P^*_h, and the sequence of lengths $n^{x,y}$ of the sublists. The

[8] In this section, we will show that $M(T, \Sigma, h) \geq \sigma^{h+1}$.

partition is easily stored using a bitvector of length σ^h (or a subset encoding of the partition using $\lceil \lg \binom{\sigma^h}{|P_h^*|} \rceil \leq \sigma^h$ bits). To store the sequence of lengths $n^{x,y}$, we simply store the concatenation the gamma codes for each length $n^{x,y}$ and bound its length.

We briefly review Elias' gamma and delta codes [Eli75] before detailing the proof. The gamma code for a positive integer ℓ represents ℓ in two parts: the first encodes $1 + \lfloor \lg \ell \rfloor$ in unary, followed by the value of $\ell - 2^{\lfloor \lg \ell \rfloor}$ encoded in binary, for a total of $1 + 2\lfloor \lg \ell \rfloor$ bits. For example, the gamma codes for $\ell = 1, 2, 3, 4, 5, \ldots$ are $\mathbf{1}, \mathbf{01\,0}, \mathbf{01\,1}, \mathbf{001\,00}, \mathbf{001\,01}, \ldots$, respectively. The delta code requires fewer bits asymptotically by encoding $1 + \lfloor \lg \ell \rfloor$ via the gamma code rather than in unary. For example, the delta codes for $\ell = 1, 2, 3, 4, 5, \ldots$ are $\mathbf{1}, \mathbf{010\,0}, \mathbf{010\,1}, \mathbf{011\,00}, \mathbf{011\,01}, \ldots$, and require $1 + \lfloor \lg \ell \rfloor + 2\lfloor \lg \lg 2\ell \rfloor$ bits. Now, we describe a simple upper bound on encoding the empirical statistical model.

Lemma 9. *The empirical statistical model for a text T drawn from an alphabet Σ can be encoded using at most $M(T, \Sigma, h) = O\left(\sigma^{h+1} \lg(1 + n/\sigma^{h+1})\right)$ bits of space, where h is the context length.*

Proof. In this encoding, we represent the lengths using the gamma code. We obtain a bitvector Z by concatenating the gamma codes for $n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}$ for $x = 1, 2, \ldots, |P_h^*|$. The bitvector Z contains $O(\sum_{x \in P_h^*, y \in \Sigma} \lg n^{x,y})$ bits; this space is maximized when all lengths $n^{x,y}$ are equal to $\Theta(n/(|P_h^*| \times \sigma) + 1)$ by Jensen's inequality [CT91]. Since $|P_h^*| \leq \sigma^h$, we bound the total space by $O\big((|P_h^*| \times \sigma) \lg\big(1 + n/(|P_h^*| \times \sigma)\big)\big) = O\big(\sigma^{h+1} \lg(1 + n/\sigma^{h+1})\big)$ bits. We do not need to encode n as it can be recovered from the sum of the sublists lengths. $\square$

The result of Lemma 9 is interesting, but it carries a dependence on n, unlike the bounds in related work, which are related only to σ and h [FGMS05]. In the next section, we show an alternate analysis that remedies this problem and relates the encoding costs to the modified entropy nH_h^*, as defined in Section 2.2.1.

2.5.2 Nearly Tight Upper Bound on $\boldsymbol{M(T, \Sigma, h)}$

In this section, we describe a nearly tight upper bound for encoding the empirical statistical model. As we described in Section 2.5.1, we can easily store the partition of Σ^h induced by the optimal prefix cover P_h^* using at most σ^h bits. We provide a new analysis for storing the sequence of lengths $n^{x,y}$ in Theorem 4.

Theorem 4. *The empirical statistical model for a text T drawn from an alphabet Σ requires at most $M(T, \Sigma, h) \leq nH_h^* + \lg n + O\left(\sigma^{h+1} \lg \sigma^{h+1}\right)$ bits of space.*

The results of Theorem 4 highlight a remarkable property of the Burrows-Wheeler Transform, namely that maintaining the statistics of the text requires *more* space than the actual encoding of the information.

To prove Theorem 4, we have to encode the sublist lengths $n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}$, where $x = 1, 2, \ldots, |P_h^*|$ and $\sum_{x \in P_h^*, y \in \Sigma} n^{x,y} = n$. We use the following encoding scheme for each context x:

- If context x contains a single nonempty sublist y, we use σ bits to mark the yth sublist as nonempty. Then, we store the length $n^{x,y} = n^x$.
- If context x contains two or more nonempty sublists, we again use σ bits to mark the nonempty sublists. To describe the rest of the method, let $n'_1 = n^x$ and $n'_j = n^x - \sum_{i=1}^{j-1} n^{x,i}$ be a scaled universe where $j \geq 2$. We use σ bits for context x, one bit per sublist. The bit for sublist y is $\mathbf{1}$ if and only if $n^{x,y} > n'_y/2$; in this case, we set $t_y = n'_y - n^{x,y}$. Otherwise, we set the bit for sublist y to $\mathbf{0}$ and set $t_y = n^{x,y}$. Notice that $t_y \leq n'_y/2$ in both cases. Now, we encode t using its delta code. Given n'_y and t_y, we can recover the value of $n^{x,y}$ as expected.

Lemma 10. *We can encode the sublist lengths $n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}$ for any context x with two or more nonempty sublists using at most $\gamma n^x H_0^*(w_x) + O(\sigma)$ bits, where $0 < \gamma < 1/2$ is a constant.*

Proof. Our scheme requires 2σ bits to store auxiliary information. Now we bound the total size of encoding the values $t_1, t_2, \ldots, t_\sigma$ using the delta code for each nonempty sublist. Our approach is to amortize the cost of writing the delta code of t_y with the encoding of its associated sublist y. We introduce some terminology to clarify the proof. For any arbitrarily fixed constant γ with $0 < \gamma < 1/2$, let $t_\gamma > 0$ be constants such that for any integer $t > t_\gamma$, $\lg t + 2 \lg\lg(2t) + 1 < \gamma(2t - \lg t - 1)$.

Then, for nonempty sublists y with $t_y \leq t_\gamma$, the delta code for t_y will take $O(\lg t_\gamma) = O(1)$ bits of space. Summing these costs for all such sublists, we would require at most $O(\sigma \lg t_\gamma) = O(\sigma)$ bits for context x.

For nonempty sublists y with $t_y > t_\gamma$, we use at most $\gamma(2t_y - \lg t_y - 1)$ bits to write the delta code of t_y using the observation above.

Now, we will use the fact that $t_y \leq n'_y/2$ for each sublist y in our scheme to bound the encoding length of sublist y, and then amortize accordingly. In general, for any $1 < t < n/2$, $\lg \binom{n}{t} \geq \lg \binom{2t}{t} \geq \lg(2^{2t}/2t) = 2t - \lg t - 1$. Since each sublist y with $t_y > t_\gamma$ satisfies this condition by the construction of our scheme, we can bound the delta code of t_y by $\gamma \lg \binom{n'_y}{t_y}$ bits. Summing over all such sublists for context x, we would require at most $\gamma \lg \binom{n^x}{n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}} + \sigma = \gamma n^x H_0^*(w_x) + \sigma$ bits using the analysis from Section 2.4.2, thus proving the lemma. $\square$

The above scheme requires us to store the length n^x of each context x, since the sum of the t_y values we store may be less than n^x. (For the case with a single nonempty context, n^x is the size of the only sublist.) For example, suppose for some context x, $n^x = 20$, $n^{x,1} = 11$, $n^{x,2} = 3$, $n^{x,3} = 5$, $n^{x,4} = 1$. According to our scheme, we would store the t_y values 9, 3, 1, and 1, which sum up to $14 < 20$. To determine the value of $n^{x,1}$, we must therefore compute $n^x - t$; thus, we must know the value of n^x.

The storage of n^x is a subtle but important point, and it is a key component in understanding the *lower bound* on encoding length for the BWT, which we discuss more in Section 2.6. In Lemma 11, we describe a technique to store the sequence of lengths n^x for $x = 1, 2, \ldots, |P_h^*|$ succinctly.

Lemma 11. *The sequence of lengths n^x for $x = 1, 2, \ldots, |P_h^*|$ can be stored using*

$$\sum_{x \in P_h^*} \lg n^x + \lg n + O(\sigma^{h+1} \lg \sigma^{h+1})$$

bits of space.

Proof. For each context x with n^x entries, we encode its length n^x in binary using $b(x) = \lfloor \lg n^x \rfloor + 1$ bits. These $b(x)$ bits do not permit a decoding of n^x by themselves, since they are not prefix codes. We describe how to fix this problem. We permute the contexts x so that they are sorted by their $b(x)$ values. Now, contexts requiring the same number of bits $b(x)$ to store their lengths are contiguous. In other words, we know that for any two consecutive contexts x and x' in the sorted order, either $b(x) = b(x')$ or $b(x) < b(x')$. What remains is the storage of the positions where $b(x) < b(x')$. We store this information using $|P_h^*|$ bits.

To remember which lengths $b(x)$ actually occur, we observe that the number of distinct lengths is at most $\lg n + 1$, since $1 \leq b(x) \leq \lg n + 1$. We store a bitvector of length $\lg n + 1$ bits to keep track of this information. Finally, we store the permutation to restore the original order of the contexts using $O(\lg |P_h^*|!) = O(\sigma^{h+1} \lg \sigma^{h+1})$ bits, thus proving the lemma. □

The above lemma allows us to store the length of each context x in $\lg n^x$ bits, plus some small additional costs. We can bound the space of our encoding scheme with the following lemma.

Lemma 12. *The empirical statistical model for a text T drawn from alphabet Σ requires at most $(1 + \gamma)nH_h^* + \lg n + O(\sigma^{h+1} \lg \sigma^{h+1})$ for all contexts x, where $0 < \gamma < 1/2$ is a constant.*

Proof. Using the definition of modified hth-order empirical entropy in Equation (2.7), we bound the first term in Lemma 11 by $\sum_{x \in P_h^*} \lg n^x \leq nH_h^*$. According to our scheme, storing the length n^x along with σ bits is sufficient to encode any context x with a single nonempty sublist. For the remaining contexts, we apply Lemma 10 to achieve the desired result. $\square$

We can further improve our bound by amortizing the cost of storing the length n^x for context x with the encoding of its sublists. The technique is reminiscent of the one we used in Lemma 10. We change our encoding scheme as follows.

- If context x contains a single nonempty sublist y, we use σ bits to mark the yth sublist as nonempty. Then, we store the length $n^{x,y} = n^x$.
- If context x contains two or more nonempty sublists, we use the scheme below.
- Let t_γ be defined as in Lemma 10. For any arbitrarily fixed constant γ with $0 < \gamma < 1/2$, let $n_\gamma > 0$ be a constant such that for any integer $n > n_\gamma$ and $t > t_\gamma$ with $t \leq n/2$, $\binom{n}{t} \geq \frac{n(n-1)\ldots(n-\lceil 1/\gamma \rceil)}{(\lceil 1/\gamma \rceil + 1)!} > n^{\lceil 1/\gamma \rceil}$.
- Instead of encoding $n^{x,1}$ as the first sublist length for context x, we use σ bits to indicate that we encode n^{x,y_b} first, where $t_b = \min\{n^{x,y_b}, n^x - n^{x,y_b}\}$ satisfies the condition $\binom{n^x}{t_b} > (n^x)^{\gamma^{-1}}$. If no such y_b exists, encode $n^{x,1}$ as before.

Lemma 13. *We can encode the sublist lengths $n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}$ along with the context length n^x for any context x with two or more nonempty sublists using at most $n^x H_0^*(w_x) + O(\sigma)$ bits.*

Proof. The cost for encoding the sublist lengths is analyzed using Lemma 10. We focus on bounding the cost for $\lg n^x$. If any sublist y_b satisfies the constraint in our scheme, we know that $\lg n^x < \gamma \lg \binom{n^x}{t_b}$, which is the same upper bound on the number of bits required to encode t_b. Thus, encoding both n^{x,y_b} and n^x will take $2\gamma \lg \binom{n^x}{t_b} + O(\sigma)$ bits of space. The encoding size for the rest of the new sequence remains the same as we observed in Section 2.4.2, thus we require at most $2\gamma n^x H_0^*(w_x) + O(\sigma)$ bits. Since $\gamma < 1/2$, this shows the bound for contexts x that satisfy the constraint.

If no sublist satisfied the constraint, then we know that each $t_i \leq t_\gamma (1 \leq i \leq \sigma)$ so the delta code for each t_i takes $O(\lg t_\gamma) = O(1)$ bits, which take at most $O(\sigma)$ bits overall. Then, the $\lg n^x$ bits for encoding n^x can be bounded by $n^x H_0^*(w_x)$ as in Lemma 12, since $n^x H_0^*(w_x) \geq \lg n^x$. This case will contribute at most $n^x H_0^*(w_x) + O(\sigma)$ bits to the bound, thus proving the bound. □

Combining Lemma 13 with our scheme for encoding the singleton context, we prove Theorem 4.

2.6 Nearly Tight Lower Bounds for the BWT

Manzini conjectures that the BWT cannot be compressed to just $nH_h^* + \lg n + g_h$ bits of space, where $g_h = O(\sigma^{h+1} \lg \sigma)$. However, in Section 2.5.2, we provide an analysis that gives an upper bound of $n(H_h + H_h^*) + \lg n + g_h''$ bits, where $g_h'' = O(\sigma^{h+1} \lg \sigma^{h+1})$. Since there are an infinite number of texts where $nH_h = 0$ but $nH_h^* \neq 0$, our bound is $nH_h + M(T, \Sigma, h) \leq nH_h^* + \lg n + g_h''$ in these cases, matching Manzini's conjectured lower bound (but not for all texts).

The BWT has been analyzed extensively since its original introduction in 1994, especially in the information-theory community [Ris84, WMF94, EVKV02]. These results apply to a wide range of statistical models for generating a text T, including high-order Markov sources, tree sources, and finite-state machine (FSM) sources. In a text indexing setting, recent theoretical results [Man01, FGMS05, KLV06] have shed light on the success of the BWT and present some limits on its compressibility. Within the text indexing framework, we will explore other classes of texts that help establish a non-trivial lower bound on the compressibility of the BWT. Surprisingly, the encoding of the BWT requires an amount of space very close to our encoding length for the upper bound. In particular, we will prove the following theorem, which shows that our upper bound analysis is nearly tight.

Theorem 5. *For any chosen positive constant $\delta \leq 1$ and fixed positive integer $k = O(\text{polylg}(n)) > \lceil 1/\delta \rceil$, there exists an infinite family of texts such that for any texts of length n in the family, its bound in Formula (1.1) satisfies the following two relations:*

$$nH_h^* + \left(\frac{k-1}{k}\right)\delta nH_h - O(\text{poly}(kd)) \leq nH_h + M(T, \Sigma, h) \tag{2.18}$$

and

$$nH_h + M(T, \Sigma, h) \leq nH_h^* + \delta nH_h + \lg n + g_h''. \tag{2.19}$$

When $\delta > 1$, we use Formula (2.1) as the upper bound for (2.19). To prove Inequality (2.19), we give a tighter analysis of the space-intensive part of the encoding scheme from Section 2.5. To capture the primary challenge from Section 2.5, we define a *δ-resilient* text. Let BWT(T) denote the result of applying the BWT to the text T. For any given constant δ such that $0 < \delta \leq 1$, the text T is *δ-resilient* if the optimal partition induced by P_h^* for BWT(T) satisfies $\max_{y \in \Sigma}\{n^{x,y}\} \leq n^x - \lceil 1/\delta \rceil$ for every context $x \in P_h^*$. In other words, no partition x of BWT(T) induced by P_h^* contains more than $n^x - \lceil 1/\delta \rceil$ identical symbols. We define $d = \lceil 1/\delta \rceil$. Now, we apply Theorem 4 to δ-resilient texts and achieve the following lemma.

Lemma 14. *For any constant δ with $0 < \delta \leq 1$ and any δ-resilient text T of n symbols over Σ, we have $nH_h + M(T, \Sigma, h) \leq n(\delta H_h + H_h^*) + \lg n + g_h''$.*

Proof. Let $\delta = 2\gamma$, where γ is chosen as in Section 2.5.2. In the proof of Theorem 4, all cases contribute at most δnH_h^* bits to the bound, except for the last case in Lemma 13. In this case, for context x, $t_y \leq t_\gamma = O(1)$ for all sublists y. Since T is δ-resilient, the largest sublist y' in context x contains $n^{x,y'} \leq n^x - \lceil 1/\delta \rceil$ entries, while the other $O(1)$ sublists consist of $\lceil 1/\delta \rceil \leq \sum_{y \neq y'} n^{x,y} \leq t_\delta = O(1)$ entries. The sublist encoding for context x requires $\lg \binom{n^x}{n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}} \geq \lceil 1/\delta \rceil \lg n^x + O(1)$ bits. To encode the empirical statistical model, we write the value of n^x in $\lg n^x$ bits using Lemma 11 and the values of $n^{x,y}$ for $y \neq y'$ in $O(1)$ bits overall (just like we did in Theorem 4). Hence, the contribution of encoding this information for $M(T, \Sigma, h)$ is $\lg n^x + O(1) \leq \delta \lg \binom{n^x}{n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}} + O(1)$ bits. Since $H_h^* \geq H_h$ (Section 2.2) and the rest of the proof is identical to the proof of Theorem 4, our lemma is proved. $\square$

To prove our lower bound Inequality (2.18) from Theorem 5, we take the following steps.

- We describe a construction scheme that takes user-defined parameters and creates a δ-resilient text T of length n.
- We count the total number of δ-resilient texts that our construction scheme generates, and use a combinatorial argument to bound the space required to distinguish between these texts.
- To achieve an entropy bound, we take an arbitrary δ-resilient text T and show that Inequality (2.18) holds.

2.6.1 Constructing δ-resilient Texts

In this section, we describe how to construct δ-resilient texts using a generalized construction scheme; then, we will use the resulting class of texts to prove Inequality (2.18) of Theorem 5. First, we define some terminology that will help clarify the discussion. Let $d = \lceil 1/\delta \rceil$, where $0 < \delta \leq 1$ is a constant. Let T_s be a support text composed of an alphabet $\Sigma = \{\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_k, \mathtt{b}, \mathtt{c}_1, \mathtt{c}_2, \ldots, \mathtt{c}_k, \mathtt{\#}\}$ of length n_s, where $k = O(\mathrm{polylg}(n)) > d$ is a fixed positive integer. Without loss of generality, we assume that $\mathtt{a}_i < \mathtt{a}_{i+1} < \mathtt{b} < \mathtt{c}_j < \mathtt{c}_{j+1} < \#$ for all i and j. We define T_s as

$$T_s = \underbrace{(\mathtt{a}_1\mathtt{c}_1)^{\ell_1}}_{r_1}\ \underbrace{(\mathtt{a}_2\mathtt{c}_2)^{\ell_2}}_{r_2}\ \dots\ \underbrace{(\mathtt{a}_k\mathtt{c}_k)^{\ell_k}}_{r_k},$$

where each $\ell_i \geq d$. We define a run r_i as the sequence of ℓ_i substrings of the form $\mathtt{a}_i\mathtt{b}^*\mathtt{c}_i$. In T_s, $\mathtt{b}$ never appears. The length of the support text T_s is $n_s = 2\sum_{i=1}^{k} \ell_i$. Consider the support text $T_s = \mathtt{a}_1\mathtt{c}_1\mathtt{a}_1\mathtt{c}_1\mathtt{a}_2\mathtt{c}_2\mathtt{a}_2\mathtt{c}_2\mathtt{a}_2\mathtt{c}_2\mathtt{a}_3\mathtt{c}_3\mathtt{a}_3\mathtt{c}_3\mathtt{a}_3\mathtt{c}_3\mathtt{a}_3\mathtt{c}_3$. Here, $k = 3$, $\ell_1 = 2$, $\ell_2 = 3$, and $\ell_3 = 4$. We now prove the following lemma.

Lemma 15. *The Burrows-Wheeler transform of the support text T_s is $\text{BWT}(T_s)$ is*

$$\text{BWT}(T_s) = \overbrace{\underbrace{\mathtt{c}_k(\mathtt{c}_1)^{\ell_1-1}}_{P_1}\ \underbrace{\mathtt{c}_1(\mathtt{c}_2)^{\ell_2-1}}_{P_2}\ \dots\ \underbrace{\mathtt{c}_{k-1}(\mathtt{c}_k)^{\ell_k-1}}_{P_k}}^{B_1}\ \overbrace{\underbrace{(\mathtt{a}_1)^{\ell_1}}_{Q_1}\ \underbrace{(\mathtt{a}_2)^{\ell_2}}_{Q_2}\ \dots\ \underbrace{(\mathtt{a}_k)^{\ell_k}}_{Q_k}}^{B_2},$$

where $B_1 = P_1P_2 \dots P_k$ is the first block of the BWT *transform, and $B_2 = Q_1Q_2 \dots Q_k$ is the second block. Here, P_i refers to the positions of the* BWT *corresponding to strings that start with symbol $\mathtt{a}_i$, and Q_i refers to positions of the* BWT *corresponding to strings that start with symbol $\mathtt{c}_i$.*

Proof. Consider the strings in the BWT matrix M, sorted in lexicographical order. According to the rank of symbols in alphabet Σ, all strings beginning with $\mathtt{a}_i$ will precede strings before $\mathtt{a}_{i+1}$. Similarly, strings beginning with $\mathtt{c}_i$ will precede strings beginning with $\mathtt{c}_{i+1}$. Finally, all strings beginning with $\mathtt{a}_i$ will precede strings beginning with $\mathtt{c}_1$. Also, there are exactly ℓ_i strings that begin with $\mathtt{a}_i$ and $\mathtt{c}_i$. We now focus on the strings that begin with $\mathtt{c}_i$.

Each string beginning with $\mathtt{c}_i$ has the symbol $\mathtt{a}_i$ preceding it (or equivalently, at the end of the string) in all cases. Thus, the part of the BWT corresponding to strings beginning with $\mathtt{c}_i$ is $(\mathtt{a}_i)^{\ell_i}$. Collectively, we call this block B_2.

Each string beginning with $\mathtt{a}_i$ has the symbol $\mathtt{c}_i$ preceding it (or at the end of the string, since it's cyclic), except the string corresponding to the first $\mathtt{a}_i$ in run r_i. This string is lexicographically the first string among all of the strings beginning with $\mathtt{a}_i$ and is preceded by $\mathtt{c}_{i-1}$ or $\mathtt{c}_k$ if $i = 1$. Thus, the part of the BWT corresponding to strings beginning with $\mathtt{a}_i$ is $\mathtt{c}_{i-1}(\mathtt{c}_i)^{\ell_i - 1}$. If $i = 1$, $\mathtt{c}_{i-1}$ is replaced with $\mathtt{c}_k$. Collectively, we call this block B_1.

Thus, the lemma is proved. □

For our example support string $T_s = \mathtt{a}_1\mathtt{c}_1\mathtt{a}_1\mathtt{c}_1\mathtt{a}_2\mathtt{c}_2\mathtt{a}_2\mathtt{c}_2\mathtt{a}_2\mathtt{c}_2\mathtt{a}_3\mathtt{c}_3\mathtt{a}_3\mathtt{c}_3\mathtt{a}_3\mathtt{c}_3\mathtt{a}_3\mathtt{c}_3$, the resulting BWT is $\mathrm{BWT}(T_s) = \overbrace{\underbrace{\mathtt{c}_3\mathtt{c}_1}_{P_1}\ \underbrace{\mathtt{c}_1\mathtt{c}_2\mathtt{c}_2}_{P_2}\ \underbrace{\mathtt{c}_2\mathtt{c}_3\mathtt{c}_3\mathtt{c}_3}_{P_3}}^{B_1}\ \overbrace{\underbrace{\mathtt{a}_1\mathtt{a}_1}_{Q_1}\ \underbrace{\mathtt{a}_2\mathtt{a}_2\mathtt{a}_2}_{Q_2}\ \underbrace{\mathtt{a}_3\mathtt{a}_3\mathtt{a}_3\mathtt{a}_3}_{Q_3}}^{B_2}$.

Now, we introduce $d = \lceil 1/\delta \rceil$ partition vectors $v_i = \langle v_i[1], v_i[2], \ldots v_i[k] \rangle$ that will generate a δ-resilient property for B_2; B_1 remains unchanged, but will implicitly encode the length of the corresponding portions of B_2. In particular, we augment T_s as follows: for each entry of v_i for all i, we replace the $v_i[j]$th occurrence of the string $\mathtt{a}_j\mathtt{c}_j$ with $\mathtt{a}_j\mathtt{b}\mathtt{c}_j$. We will make d such replacements in each of the k partitions. We call this augmented text T'_s, of length $n'_s = n_s + dk$.

Lemma 16. *The Burrows-Wheeler transform of the augmented text T'_s is* $\mathrm{BWT}(T'_s)$ *is*

$$\mathrm{BWT}(T'_s) = \overbrace{P'_1 P'_2 \ldots P'_k}^{B_1}\ \overbrace{(\mathtt{a}_1)^d (\mathtt{a}_2)^d \ldots (\mathtt{a}_k)^d}^{A}\ \overbrace{Q'_1 Q'_2 \ldots Q'_k}^{B_2},$$

where P'_i is composed of symbols preceding strings that start with $\mathtt{a}_i$, A is composed of symbols preceding strings that start with $\mathtt{b}$, *and Q'_i is composed of symbols preceding strings that start with $\mathtt{c}_i$.*

Proof. This proof is similar to Lemma 15, where each string in P_i precedes strings in P_{i+1}. Here, all strings in P'_i precede strings in P'_{i+1}, strings in Q'_i precede strings in Q'_{i+1}, and strings in P'_i precede strings beginning with $\mathtt{b}$ (called A) and strings in A precede strings in Q'_1.

Then, P'_i is a string of length ℓ_i similar to P_i, but the single occurrence of $\mathtt{c}_{i-1}$ (or $\mathtt{c}_k$ if $i = 1$) could be in any of the ℓ_i positions. Also, Q'_i is a string of length ℓ_i where

d positions contain the symbol $\mathtt{b}$, and all others are $\mathtt{a}_i$. Block A consists of exactly d occurrences of each $\mathtt{a}_i$ sorted in lexicographical order, since all d strings beginning with $\mathtt{bc}_i$ precede all strings beginning with $\mathtt{bc}_{i+1}$, thus finishing the proof. $\square$

Consider the augmented string $T'_s = \mathtt{a_1bc_1a_1bc_1a_2c_2a_2bc_2a_2bc_2a_3bc_3a_3c_3a_3bc_3a_3c_3}$, where $d = 2$. Then, $\textsc{bwt}(T'_s) = \overbrace{\underbrace{\mathtt{c_3c_1}}_{P'_1}\ \underbrace{\mathtt{c_2c_2c_1}}_{P'_2}\ \underbrace{\mathtt{c_3c_2c_3c_3}}_{P'_3}}^{B_1}\ \overbrace{\mathtt{a_1a_1a_2a_2a_3a_3}}^{A}\ \overbrace{\underbrace{\mathtt{bb}}_{Q'_1}\ \underbrace{\mathtt{a_2bb}}_{Q'_2}\ \underbrace{\mathtt{a_3a_3bb}}_{Q'_3}}^{B_2}.$

A simple verification will show that blocks A and B_2 are δ-resilient portions of T'_s. Furthermore, block A is deterministic once the parameters d and k have been chosen; block B_1 encodes the length of each Q'_i. To have a fully δ-resilient text, we want B_1 to have the same property, so we generate the string $T = T'_s(T_s)^{d-1}\mathtt{\#}$. This will include $d-1$ occurrences of a different symbol inside each P'_1. Note that $|T| = n = dn_s + dk + 1$.

Lemma 17. *Let $T = T'_s(T_s)^{d-1}\mathtt{\#}$, where T_s is the support text and T'_s is the augmented text. Then, the $\textsc{bwt}(T)$ is*

$$\textsc{bwt}(T) = \overbrace{P''_1 P''_2 \dots P''_k}^{B_1}\ A\ \overbrace{Q''_1 Q''_2 \dots Q''_k}^{B_2}\ \mathtt{c}_k,$$

where P''_i is composed of symbols preceding strings that start with $\mathtt{a}_i$, A is composed of symbols preceding strings that start with $\mathtt{b}$, and Q''_i is composed of symbols preceding strings that start with $\mathtt{c}_i$.

Proof. The strings P''_i and Q''_i are of length $d\ell_i$. Similar to the arguments in Lemma 16, P''_1 consists of the symbol $\mathtt{c}_1$ in all but $d\ell_1 - d$ positions; one positions contains $\mathtt{\#}$ and the other $d-1$ positions contain $\mathtt{c}_k$. P''_i consists of the symbol $\mathtt{c}_i$ in all but $d\ell_i - d$ positions; the other d positions contain $\mathtt{c}_{i-1}$. Each Q''_i is similar to the previous case, except its length is now $d\ell_i$. Q''_i still contains only d occurrences of $\mathtt{b}$.

Finally, the last $\mathtt{c}_k$ is the symbol preceding $\mathtt{\#}$ in the text, which is lexicographically the largest symbol, and therefore the last string represented in the $\textsc{bwt}$, thus finishing the proof. $\square$

For our example, let

$$\begin{aligned} T = T_s'T_s\mathtt{\#} &= \mathtt{a_1bc_1a_1bc_1a_2c_2a_2bc_2a_2bc_2a_3bc_3a_3c_3a_3bc_3a_3c_3} \\ & \mathtt{a_1c_1a_1c_1a_2c_2a_2c_2a_2c_2a_3c_3a_3c_3a_3c_3a_3c_3\#}. \end{aligned}$$

Then, the BWT(T_s') is

$$\begin{aligned} \text{BWT}(T_s') &= \overbrace{\underbrace{\mathtt{\#c_1c_3c_1}}_{P_1''}\ \underbrace{\mathtt{c_2c_2c_1c_1c_2c_2}}_{P_2''}\ \underbrace{\mathtt{c_3c_2c_3c_3c_2c_3c_3c_3}}_{P_3''}}^{B_1}\ \overbrace{\mathtt{a_1a_1a_2a_2a_3a_3}}^{A} \\ & \overbrace{\underbrace{\mathtt{ba_1ba_1}}_{Q_1''}\ \underbrace{\mathtt{a_2ba_2a_2ba_2}}_{Q_2''}\ \underbrace{\mathtt{a_3a_3bba_3a_3a_3a_3}}_{Q_3''}}^{B_2}\,\mathtt{c_3}. \end{aligned}$$

Now we analyze the cost of encoding a δ-resilient text.

2.6.2 Encoding a δ-resilient Text

In this section, we analyze the space required to store a δ-resilient text. Since B_1 and A are deterministic once d and k are chosen, we focus only on the encoding cost of B_2. First, we prove the following lemma.

Lemma 18. *For any set of p objects, at least half of them will take at least* $\lg p - 1$ *bits to encode so that the objects can be distinguished from one another.*

Proof. Since one can distinguish at most 2^j objects from one another using j bits, the most succinct encoding would greedily store two objects using one bit each, four objects using two bits each, and so on. Thus, we need to make sure that $\sum_i^j 2^i \geq p$. Thus, $j + 1 \geq \lg p$, and the lemma follows. □

Let Λ be the set of all possible choices λ of length parameters $\ell_1, \ell_2, \ldots, \ell_k$ used to generate δ-resilient texts in Section 2.6.1. By construction, $|\Lambda| = \binom{n_s/2-dk+k-1}{k-1}$. For a given choice λ of parameters, we choose d positions in each partition Q''_i that will contain a `b`. However, we are only choosing from the first ℓ_i positions for each run r_i (i.e., the positions that correspond to the entries in T'_s). Once these positions are chosen, we perform the steps described in our construction scheme. Since the BWT is a reversible transform, we have $\binom{\ell_i}{d}$ possible partitions Q''_i and our construction scheme generates one of

$$X = \sum_{\lambda \in \Lambda} \binom{\ell_1}{d}\binom{\ell_2}{d}\cdots\binom{\ell_k}{d}$$

different texts. We let an adversary encode the X texts in any way he wishes. Then, we use Lemma 18 to consider only half of these texts, namely the ones that take at least $\lg X - 1$ bits to encode. Now we analyze the quantity $\lg X - 1$.

To help analyze $\lg X - 1$, we divide Λ into two sets Y and Z of equal cardinality, such that for any texts $y \in Y$ and $z \in Z$, the product $p(y) \leq p(z)$, where $p(T) = \prod_1^k \binom{\ell_i}{d}$. In words, Y contains the texts T where $p(T)$ is smaller, and Z contains the ones where $p(T)$ is larger. We take a single arbitrary text S from set Y and determine which choice λ^* of length parameters ℓ_i were used. We separate the k terms corresponding to λ^* from $\lg X - 1$ and analyze their cost separately. The terms are $\sum_1^k \lg \binom{\ell_i}{d} = nH'_1(S)$, by our definition of finite set empirical entropy. Since $nH'_h(S) \leq nH'_1(S)$, the contribution of this part of $\lg X - 1$ is at least $nH'_h(S)$ bits. We translate this into a bound in terms of nH^*_h using the following lemma.

Lemma 19. *For a δ-resilient text, $nH^*_h - \Theta(k \lg d) \leq nH'_h$.*

Proof. It suffices to show that $nH^*_0 - \Theta(\lg d) \leq nH'_0$ for each partition Q''_i in a δ-resilient text, since there are at most $2k+1$ partitions. We apply Stirling's double inequality to the expression $\lg \binom{\ell_i}{d}$ and find that

$$\begin{aligned} \lg \binom{l_i}{d} &> \ell_i H_0 + \frac{1}{2} \lg \frac{\ell_i}{d(\ell_i - d)} - O(1) \\ &> \ell_i H_0 + \frac{1}{2} \lg \frac{1}{d} - O(1), \end{aligned}$$

thus proving the lemma. □

Thus, the total contribution of the part of $\lg X - 1$ corresponding to the text S is at least $nH_h^*(S) - \Theta(k \lg d)$ bits. Now we bound the term X to figure out the entire cost of encoding the string S. We will lower bound X by the sum for just the set Z and obtain

$$\begin{aligned} X &\geq \sum_{z \in Z} \prod_{1}^{k} \binom{\ell_i}{d} \\ &\geq \sum_{z \in Z} p(S) \\ &= \frac{1}{2} \binom{n_s/2 - dk + k - 1}{k-1} p(S). \end{aligned}$$

Taking logs, we require $nH_h^*(S) + \lg \binom{n_s/2-dk+k-1}{k-1} - 1$ bits of space.

To finish the proof, we analyze the contribution of the term $\lg \binom{n_s/2-dk+k-1}{k-1}$. For ease of notation, let $g = n_s/2 - dk + k - 1$. We want to show that $(k-1)\lg(g/(k-1)) \leq \lg \binom{g}{k-1}$. The claim is true by inspection when $g \leq 4$ or $k-1$ is $0, 1$, or $g-1$. For the remainder of the cases, we apply Stirling's inequality as in Theorem 1 to verify the claim. Now, $(k-1)\lg(g/(k-1)) \geq (k-1)\lg(n_s/2) - (k-1)\lg(dk) - (k-1)\lg k$. Thus, the contribution of this part of $\lg X - 1$ is at least $(k-1)\lg n - \Theta(k \lg(dk))$ bits, proving Inequality (2.18) and Theorem 5 for any arbitrary δ-resilient text S.

2.7 Random Access to the Compressed Representation of *LF* and Φ

In Section 2.3, we have described the importance of the LF mapping and the Φ function for compressing the BWT. As we shall see, these functions are also essential to performing compressed text indexing. However, we need more functionality since we need random access to their compressed values with a small cost for decoding. With the techniques discussed so far, computing the ith value of LF or Φ, for $1 \leq i \leq n$, has two major drawbacks:

- We need to decompress all the information, even if we need a single value of LF or Φ.
- The decompression is sequentially performed even though the required access is random.

We circumvent the two drawbacks above by using succinct dictionaries and compressed directories for speeding up the access and avoiding to decompress all the data while keeping the space occupancy entropy-bound. The main contribution of this section is to show how to store LF and Φ in compressed format so that each call decompresses just a small portion of their format:

- Each call takes $O(\lg \sigma)$ time using further $O(n \lg \lg n / \lg_\sigma n) = o(n \lg \sigma)$ bits of space for storing the compressed auxiliary data structures.
- Each call takes $O(1)$ time using further $O(n)$ bits for the compressed auxiliary data structures (i.e. $o(n \lg \sigma)$ bits when σ is not a constant).

We proceed in the rest of the section as follows. In Section 2.7.1, we describe how to extend the functionalities of the wavelet trees to succinct dictionaries. We then show how to use wavelet trees and some auxiliary data structures to get the random access to the compressed representation of Φ in Section 2.7.2 and to that of LF in Section 2.7.3.

2.7.1 Wavelet Trees as Succinct Dictionaries

Our compressed directories hinge on constant-time *rank* and *select* data structures [Jac89b, Mun96, Pag01, RRR02]. For a bitvector B of size n, the function $rank_{\mathbf{1}}(B, i)$ returns the number of **1**s in B up to (and including) position i. The function $select_{\mathbf{1}}(B, i)$ returns the position of the ith **1** in B. We can also define $rank_{\mathbf{0}}$ and *select*$_{\mathbf{0}}$ in terms of the **0**s in B. As previously mentioned in Section 2.4.2, subset encoding can implicitly represent B as a subset of the elements from $1 \ldots n$, associating each **1**, say in position j in B, with element j in the subset.[9] Letting t be the number of elements thus implicitly represented (the number of **1**s in the bitvector), we can replace bitvector B supporting *rank*$_{\mathbf{1}}$ and *select*$_{\mathbf{1}}$ with the constant-time *indexable dictionaries* developed by Raman, Raman, and Rao [RRR02], requiring $\lceil \lg \binom{n}{t} \rceil + O(t \lg\lg t / \lg t) + O(\lg\lg n)$ bits. As can be seen, the bound of subset encoding, $\lceil \lg \binom{n}{t} \rceil$, has an additional term for the fast-access directories, $O(t \lg\lg t / \lg t) + O(\lg\lg n)$. Moreover, *rank*$_{\mathbf{1}}(B, i) = -1$ if $B[i] \neq \mathbf{1}$ in indexable dictionaries. If we wish to support the full functionalities of *rank*$_{\mathbf{1}}$, *select*$_{\mathbf{1}}$, *rank*$_{\mathbf{0}}$, and *select*$_{\mathbf{0}}$, we need to use the *fully-indexable* version of their structure, called an FID.

Theorem 6 (Raman, Raman, and Rao [RRR02]). *An* FID *storing t items out of a universe of n items, requires*

$$\left\lceil \lg \binom{n}{t} \right\rceil + O\left(\frac{n \lg\lg n}{\lg n}\right)$$

bits of space. Each call to rank$_{\mathbf{1}}$*, select*$_{\mathbf{1}}$*, rank*$_{\mathbf{0}}$*, and select*$_{\mathbf{0}}$ *takes $O(1)$ time.*

Note that the additional term of $O(n \lg\lg n / \lg n)$ in Theorem 6 is related to the universe size n, instead of the subset size t. Analogously to what done with subset encoding, since $\lg \binom{n}{t} \leq n$, we will use FIDs as space-efficient replacements of bitvectors of length n with t **1**s (alternatively, with $n - t$ **0**s) supporting *rank* and *select* on both **0**s and **1**s.[10] In this way, we can successfully reuse part of the analysis given in Section 2.3.

[9]Note that ranking/unranking a subset refers to the lexicographic generation of subsets mentioned in Section 2.4.2, not to be confused with the *rank* function defined here.

[10]In this chapter, we write *rank*(i) or *select*(i) to denote the appropriate function on **1**s when there is no confusion.

Let us now consider the wavelet trees as defined in Section 2.4.3. What we obtain by replacing the subset encodings in the nodes with FIDs, is a generalization of *rank* and *select* operations from binary to σ-ary vectors. We adopt the notation introduced in Section 2.4.3, where s_y denotes the $\langle x, y\rangle$ sublist of $n^{x,y}$ entries and $1 \leq y \leq t^x$. (Recall that $t^x \leq \sigma$ is the number of nonempty sublists for context x, and, without loss of generality, the symbols from Σ for these sublists are renumbered from 1 to t^x.) Each contiguous portion of symbols of the BWT corresponding to context x is stored by a separate wavelet tree; we denote this portion by $w_x = w_x[1 \ldots n^x]$. To make the discussion a bit more general, we define two primitives, where $1 \leq y \leq t^x$ and $1 \leq i \leq n^x$:

- For each symbol y, function $rank'_y(w_x, i)$ returns the number of occurrences of y in w_x up to (and including) position i.
- For each symbol y, function $select'_y(w_x, i)$ returns the position of the ith occurrence of y in w_x.

When $w_x = B$ and $y \in \{\mathbf{0}, \mathbf{1}\}$, we obtain the classic *rank* and *select* operations on bitvectors B. Next, we show how wavelet trees can support *rank*$'$ and *select*$'$ efficiently using FIDs.

Lemma 20. *Using a wavelet tree for context x, we can encode the t^x nonempty sublists for that context in fewer than*

$$\lg \binom{n^x}{n^{x,1}, n^{x,2}, \ldots, n^{x,\sigma}} + O\left(t^x + n^x \frac{\lg \lg n^x}{\lg_{t^x} n^x}\right)$$

bits, so that rank$'$ and select$'$ take $O(\lg t^x)$ time.

To begin with, we augment our wavelet tree by replacing the t-subset encoding of [Knu05, Rus05] with the FID structure from [RRR02]. To resolve query $select'_y(w_x, i)$ on our new wavelet tree for w_x, we follow these steps.

$\triangleright$ $select'_y(w_x, i)$:

1. Set $s = s_y$.
2. If s is the left child, search for the ith $\mathbf{0}$ in s's parent dictionary: set $i = \mathit{select}_{\mathbf{0}}(i)$.
3. If s is the right child, search for the ith $\mathbf{1}$ in s's parent dictionary: set $i = \mathit{select}_{\mathbf{1}}(i)$.
4. Set $s = \mathit{parent}(s)$.
5. Recurse to step 2, unless $s = \mathit{root}$.
6. Return i as the answer to the query *select*′ in sublist s_y.

This query trivially requires $O(\lg t^x)$ time since *select* takes constant time and the depth of the wavelet tree is $O(\lg t^x)$ as shown in Lemma 6. The other query can be performed analogously.

$\triangleright$ $\mathit{rank}'_y(w_x, i)$:

1. Set $s = \mathit{root}$.
2. If s_y is a descendant of the left child, set $i = \mathit{rank}_{\mathbf{0}}(i)$ in s's dictionary.
3. If s_y is a descendant of the right child, set $i = \mathit{rank}_{\mathbf{1}}(i)$ in s's dictionary.
4. Set $s =$ the child of s that is an ancestor of leaf s_y.
5. Recurse to step 2, unless $s = s_y$.
6. Return i as the answer to the query *rank*′ in sublist s_y.

This query also requires $O(\lg t^x)$ time. The space analysis of the new wavelet tree is similar to that of the unaugmented wavelet tree in Lemma 6, except that we must sum the costs of the lower-order terms for the FIDs. Specifically, there are $O(\lg t^x)$ levels in the wavelet tree and, for each such level, there are universe sizes $u_1, u_2, \ldots, u_r$, such that $r < t^x$ and $\sum_{j=1}^{r} u_j \leq n^x$. Each FID gives an extra contribution of at most $cu_j \lg\lg u_j / \lg u_j$ bits to the analysis in Lemma 6, for a constant $c > 0$. For a given level in the wavelet tree, we claim that the additional number of bits is

$$\sum_{j=1}^{r} cu_j \lg\lg u_j / \lg u_j = O(n^x \lg\lg n^x / \lg n^x). \qquad (2.20)$$

Hence, we get a total of $O(n^x \lg\lg n^x / \lg_{t^x} n^x)$ bits of space for all the levels. In order to prove our claim (2.20), first note that there exists a constant $\kappa_0 > 1$ such that the function $f(\kappa) = \kappa \lg\lg \kappa / \lg \kappa$ is concave for any $\kappa > \kappa_0$. We then split the sum in Equation (2.20) in two parts. The first part involves the terms such that $u_j \leq \kappa_0$, giving a total contribution of $O(r)$, since κ_0 is constant with respect to n^x and r, the number of nonempty sublists in the given level of the wavelet tree. The second part involves only the terms such that $u_j > \kappa_0$, for which the concavity of $f(\kappa)$ holds. Multiplying by r/r and applying Jensen's inequality [CT91], we obtain

$$\frac{r}{r} \times \sum_{j=1}^{r} c \frac{u_j \lg\lg u_j}{\lg u_j} = O\left(r \times \frac{(\sum_{j=1}^{r} u_j / r) \lg\lg (\sum_{j=1}^{r} u_j / r)}{\lg (\sum_{j=1}^{r} u_j / r)} \right) = O\left(\frac{n^x \lg\lg (n^x / r)}{\lg (n^x / r)} \right).$$

Note the sum over the r values on all levels of the wavelet tree is $t^x - 1$ (i.e. the number of internal nodes), so that the total is $O(t^x + n^x \lg\lg n^x / \lg_{t^x} n^x)$ additional bits, thus completing the proof of Lemma 20. This term seems difficult to improve due to strong evidence from Miltersen [Mil05]. In the following, when we invoke the *rank* and *select* operations, we specify the dictionary they refer to unless this is clear from the context.

2.7.2 Random Access to the Compressed Representation of Φ

We now describe how to store, in compressed format, the Φ function described in Section 2.3.1, so as we can quickly compute any value $\Phi(i)$, for $1 \leq i \leq n$, by decompressing a small portion of the format. We employ the conceptual table $\mathcal{T}$ described in Section 2.3.2, and adopt $\mathcal{T}$'s encoding for the BWT given at the end of

Section 2.3.2, except that the wavelet trees are now augmented with FIDs as discussed in Section 2.7.1 (cf. Theorem 6). Recall that in order to support a query for $\Phi(i)$, we need to decompress the ith nonempty entry in the concatenation in column major order of the sublists in $\mathcal{T}$. (We refer to Table 2.5 for an example.) We need the following basic information: the list y containing entry $\Phi(i)$; the context x such that the $\langle x, y \rangle$ sublist contains $\Phi(i)$; the element z stored explicitly in the *normalized* $\langle x, y \rangle$ sublist (see Table 2.5); the number of elements $\#x$ in all contexts prior to x. In the example for $\Phi(2) = 10$, we have $y = \mathtt{i}$, $x = \mathtt{s}$, $\#x = 7$, and $z = 3$. The value for $\Phi(i)$ is then $\#x + z$ because of the normalization of the sublists described in Section 2.3.2. We execute five main steps to answer a query.

$\triangleright$ *Query* $\Phi(i)$:

1. Consult a directory G to determine $\Phi(i)$'s list y and the number of elements in all prior lists, $\#y$. (We now know that $\Phi(i)$ is the $(i - \#y)$th element in list y.) In the example above, we consult G to find $y = \mathtt{i}$ and $\#y = 0$.
2. Consult a list L^y to determine the context x of the $(i - \#y)$th element in list y. For example, we consult $L^{\mathtt{i}}$ to determine $x = \mathtt{s}$. We identify the $\langle x, y \rangle$ sublist and $\#p$, the number of entries in previous sublists $\langle x, y' \rangle$ with $y' < y$.
3. Look up the appropriate entry in $\langle x, y \rangle$ to find z. This entry occupies position $i - \#y - \#p$ inside $\langle x, y \rangle$; hence, $z = \mathit{select}'_y(i - \#y - \#p)$ for context x. In the example, we look for the first entry in the $\langle \mathtt{s}, \mathtt{i} \rangle$ sublist and determine $z = 3$.
4. Consult a directory F to determine $\#x$, the number of elements in all prior contexts. In the example, after looking at F, we determine $\#x = 7$.
5. Return $\#x + z$ as the solution to $\Phi(i)$. The example would then return $\Phi(i) = \#x + z = 7 + 3 = 10$.

We now detail some of the steps given above, describing the set of auxiliary data structures.

Directories G and F

We describe the details of the directory G (and the analogous structure F), which determines $\Phi(i)$'s list y and the number of elements in all prior lists $\#y$. We can think of G conceptually as a bitvector of length n. For each nonempty list y (considered in lexicographical order) containing $n^y = \sum_{x \in P_h^*} n^{x,y}$ elements (where P_h^* is the optimal prefix cover defined in Section 2.2), we write a **1**, followed by $(n^y - 1)$ **0**s. Intuitively, each **1** represents the first element of a list. Since there are as many **1**s in G as nonempty lists, G cannot have more than $l = \sigma$ **1**s. To retrieve the desired information in constant time, we compute $y = \mathit{rank}(G, i)$ and $\#y = \mathit{select}(G, y) - 1$. The F directory is similar, where each **1** denotes the start of a context x (considered in lexicographical order), rather than the start of a list, followed by $(n^x - 1)$ **0**s. Since there are at most $c = |P_h^*| \leq \sigma^h$ possible contexts, we have at most that many **1**s. We use FIDs to store these directories.

Lemma 21. *We can store G using*

$$\left\lceil \lg \binom{n}{l} \right\rceil + O\left(\frac{n \lg\lg n}{\lg n}\right) = O\left(\sigma \lg\left(1 + \frac{n}{\sigma}\right) + \frac{n \lg\lg n}{\lg n}\right)$$

bits of space, and F using space (in bits) of

$$\left\lceil \lg \binom{n}{c} \right\rceil + O\left(\frac{n \lg\lg n}{\lg n}\right) = O\left(|P_h^*| \lg\left(1 + \frac{n}{|P_h^*|}\right) + \frac{n \lg\lg n}{\lg n}\right).$$

List-Specific Directory L^y

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

The first component is a FID that produces a nonempty context number $p > 0$. In the example, the same context $x = \texttt{p}$ has $p = 1$ in list `i` while has $p = 2$ in list `p`. It also produces the number $\#p$ of items in all prior sublists. In the example, context $x = \texttt{p}$ has $\#p = 0$ in list `i`, and $\#p = 1$ in list `p`. To retrieve the desired information in constant time, we compute $p = \mathit{rank}(L^y, i - \#y)$ and $\#p = \mathit{select}(L^y, p) - 1$.

In order to save space, we actually store a single directory shared by all lists y. For each list y, we can retrieve the list's p and $\#p$ values. Conceptually, we represent this global directory L as a simple concatenation (in lexicographical order by y) 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 $\#p = \mathit{select}(p + \mathit{rank}(\mathit{start})) - \mathit{start} - 1 = \mathit{select}(\mathit{rank}(i)) - \mathit{start} - 1$. We implement L by a single FID storing s entries in a universe of size n, where $s = \sum_{x \in P_h^*} t^x$ is the number of nonempty sublists.

Lemma 22. *We can compute the local nonempty context number* p *and* $\#p$ *in constant time, and the space used (in bits) is*

$$\left\lceil \lg \binom{n}{s} \right\rceil + O\left(\frac{n \lg\lg n}{\lg n}\right) = O\left(s \lg\left(1 + \frac{n}{s}\right) + \frac{n \lg\lg n}{\lg n}\right).$$

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

Lemma 23. *We can map the local nonempty context number p to x in constant time, and the space used (in bits) is*

$$\left\lceil \lg \binom{|P_h^*|\sigma}{s} \right\rceil + O\left(\frac{(|P_h^*|\sigma) \lg\lg(|P_h^*|\sigma)}{\lg(|P_h^*|\sigma)} \right) = o\left(\sigma^{h+1}\right).$$

Time and Space Complexity

Theorem 7. *The neighbor function Φ can be represented in a compressed format for a text of n symbols over the alphabet Σ using $nH_h + O(n \lg\lg n / \lg_\sigma n) + g'_h \lg(1 + n/g'_h)$ bits of space, where $g'_h = O(\sigma^{h+1})$, so that each call to Φ takes $O(\lg \sigma)$ time.*

Proof. The space occupancy is that indicated by Theorem 2, except that Lemma 6 should be replaced by Lemma 20 plus the additional terms indicated in Lemma 21, Lemma 22 and Lemma 23, where $s = \sum_{x \in P_h^*} t^x$ is bounded by g'_h. The time cost is constant except for the wavelet tree, as stated in Lemma 20, where $t^x \leq \sigma$. ☐

Theorem 8. *The neighbor function Φ can be represented in a compressed format using $nH_h + O(n) + g'_h \lg(1 + n/g'_h)$ bits of space, so that each call to Φ takes $O(1)$ time.*

Proof. The proof is analogous to that of Theorem 7, except that for each context x, the wavelet tree is replaced by a set of t^x *indexable dictionaries* [RRR02] representing sublists $\langle x, y \rangle$ for $1 \leq y \leq t^x$ with $\lceil \lg \binom{n^x}{n^{x,y}} \rceil + O(n^{x,y} \lg\lg n^{x,y} / \lg n^{x,y})$ bits (since we only need *select* operations on them, there is no need to use an FID). When we need to perform $select'_y$ for context x, we just run the *select* operation on the indexable dictionary for $\langle x, y \rangle$. By Lemma 4, using indexable dictionaries adds a term that sums up to $O(n)$ in the bound of Theorem 7, but we only perform $O(1)$ constant-time queries to a single dictionary, in total, $O(1)$ time. This scheme may pay when σ

is not a constant, since it requires additional $O(n) = o(n \lg \sigma)$ bits of space for the auxiliary data structures. $\square$

2.7.3 Random Access to the Compressed Representation of *LF*

The machinery for the compressed representation of Φ can be reused also for the LF mapping. In [FM05], it is shown that $LF(i) = C[L[i]] + Occ(i, L[i])$ for any $1 \leq i \leq n$. Here, for any $y \in \Sigma$, vector $C[y]$ counts the number of occurrences of symbols $y' < y$ appearing in the text T, and $Occ(i, y)$ is the number of occurrences of y appearing in the first i positions of the BWT (here it is identified with L). It turns out that, given i, we can compute the context x and the list $y = L[i]$ as described for *Query* $\Phi(i)$ in Section 2.7.2. Then, we can obtain $Occ(i, y)$ as the value of $rank'_y(i - \#y - \#p) + \#y$ for context x. The following are corollaries of Theorems 7 and 8.

Corollary 1. *The LF mapping can be represented in a compressed format for a text of n symbols over the alphabet Σ using $nH_h + O(n \lg\lg n / \lg_\sigma n) + g'_h \lg(1 + n/g'_h)$ bits of space, where $g'_h = O(\sigma^{h+1})$, so that each call to LF takes $O(\lg \sigma)$ time.*

In particular, we note that in Corollary 2, we can use the indexable dictionaries since we invoke $rank(i)$ for a suitable sublist $\langle x, y \rangle$, such that $y = L[i]$, the ith symbol in the BWT. This corresponds to the weak form of *rank* supported by indexable dictionaries.

Corollary 2. *The LF mapping can be represented in a compressed format using $nH_h + O(n) + g'_h \lg(1 + n/g'_h)$ bits of space, so that each call to LF takes $O(1)$ time.*

2.8 Using the Framework for Compressed Suffix Arrays

In this section, we use the machinery developed so far to achieve text indexing, showcasing the insights we obtained in our prior investigation. In the remainder of this chapter, we will detail the results of our CSA, though analogous methods hold for the FM-index implemented with the wavelet tree. In fact, there are a whole host of methods now that use Φ or the LF mapping (see the survey in [NM06a]).

2.8.1 Compressed Suffix Arrays (CSAs)

To recap, a standard suffix array [GBS92, MM93] 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, $T\big[SA[i], n\big]$. 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. In order to achieve self-indexing, we also use the notion of the *inverse* suffix array SA^{-1}, such that $SA^{-1}[j] = i$ if and only if $SA[i] = j$. In other words, $SA^{-1}[j]$ gives the rank in the lexicographic order of suffix $T[j, n]$ among the suffixes of T.

The CSA contains the same information as a standard (inverse) suffix array, though it operates only on a compressed format. For the rest of the chapter, we assume that the CSA supports the following set of operations as given in [GV05, Sad03, Sad02b].

Definition 2. Given a text T of length n, a *compressed suffix array* (CSA) for T supports the following operations without requiring explicit storage of T or its (inverse) suffix arrays, SA and SA^{-1}:

- *compress*(T) produces a compressed representation, Z, that encodes (i) text T, (ii) its suffix array SA, and (iii) its inverse suffix array SA^{-1};

- $\textit{lookup}_Z(i)$ returns the value of $SA[i]$, the position of the ith suffix in lexicographical order, for $1 \leq i \leq n$;
- $\textit{lookup}_Z^{-1}(j)$ returns the value of $SA^{-1}[j]$, the rank of the jth suffix in T, for $1 \leq j \leq n$;
- $\textit{substring}_Z(i, c)$ decompresses the first c symbol prefix of the suffix $T\big[SA[i], n\big]$, for $1 \leq i \leq n$ and $1 \leq c \leq n - SA[i] + 1$.

We drop some of the parameters from the operations listed in Definition 2 whenever their usage is clear from the context. For example, if we wish to decompress the $c = 6$ symbols belonging to the text substring $T[18, 25]$, we indicate the corresponding operations as follows. First we find the lexicographic position, $\textit{lookup}^{-1}(18) = 16$, of its corresponding suffix and then we execute $\textit{substring}(16, c)$.

The structure of a CSA is recursive in nature, where each of the $\ell = \lg \lg_\sigma n$ levels indexes half the elements of the previous level. Hence, the kth level indexes $n_k = n/2^k$ elements. We review and use this recursive decomposition given below:[11]

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

SA_k is not explicitly stored (except at the last level ℓ), but we refer to it for the sake of explanation. B_k is a bitvector such that $B_k[i] = \mathbf{1}$ if and only if $SA_k[i]$ is even. Even-positioned suffixes are represented in SA_{k+1} with their positions divided by 2. 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

[11]We use the neighbor function Φ_k to emphasize its importance to our methods; for the full level approach, Grossi and Vitter use the partial function Ψ_k in their exposition.

the value $p+1$. In words, Φ_k is the Φ function from Section 2.3.1 applied to SA_k instead of SA. Hence, we can equivalently describe Φ_k by the following formula (also handling the case when $SA_k[i] = n$):

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

A *lookup* for $SA_k[i]$ can be answered in the following way:

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

An example of the recursion for a text T is given below, where `a` < `b` < `#` and `#` is a special end-of-text symbol. (The text T is borrowed from [GV05], but note that the Φ_k function is used instead.) No further levels are needed, since the four suffix array pointers at level 3 are stored explicitly.

	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

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*
SA_1:	8	5	2	14	12	7	6	9	3	10	15	4	1	13	11	16
B_1:	**1**	**0**	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**1**
$rank(B_1, i)$:	1	1	2	3	4	4	5	5	5	6	6	7	7	7	7	8
Φ_1:	8	7	9	11	14	1	6	10	12	15	16	2	3	4	5	13

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
SA_2:	4	1	7	6	3	5	2	8
B_2:	**1**	**0**	**0**	**1**	**0**	**0**	**1**	**1**
$rank(B_2, i)$:	1	1	1	2	2	2	3	4
Φ_2:	6	7	8	3	1	4	5	2

	1	*2*	*3*	*4*
SA_3:	2	3	1	4

Here, $\Phi_0(4) = 16$, since $SA_0[4] = 17$ and $SA_0[16] = 17+1 = 18$. For this example, suppose we already know SA_1. To retrieve $SA_0[16]$, since $B_0[16] = \mathbf{1}$, we compute $2 \cdot SA_1[\mathit{rank}(B_0, 16)] = 2 \cdot SA_1[8] = 2 \cdot 9 = 18$. 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 CSA has two incarnations that show some inherent space/time tradeoffs. The first (time-efficient) version reduces the space requirement to $O(n \lg \sigma \lg \lg_\sigma n)$ bits, while *lookup* takes only $O(\lg \lg_\sigma n)$ time. This version explicitly uses the recursive structure explained above. The second (space-efficient) version skips all but a constant fraction ϵ of these levels, for some $0 < \epsilon \leq 1$, relying on a succinct dictionary D_k to perform the task of B_k, but instead mapping elements several levels away. This scheme reduces the space requirement to $O(\epsilon^{-1} n \lg \sigma)$, however *lookup* now takes $O(\lg_\sigma^\epsilon n)$ time. In practice, the second scheme is much better, as the slowdown in searching is reasonable. We remark that Sadakane [Sad03] has shown that the space complexity can be restated in terms of the order-0 entropy $H_0 \leq \lg \sigma$, giving as a result $O(\epsilon^{-1} H_0 \, n)$ bits.

In order to compress SA^{-1} along with SA, it suffices to keep SA_ℓ^{-1} in the last level ℓ, as the rest of the machinery for compressing SA and SA^{-1} is identical [Sad03, Sad02b]. Hence the cost of *lookup*$^{-1}$ is the same as that for *lookup*, and it suffices to discuss the latter only. Moreover, it is not difficult to extend the *substring* operation using Φ_k for any value of k, such that each application of Φ_k decompresses $\Theta(2^k)$ symbols at a time, for a total cost of $O(c/2^k)$ time plus the cost of a *lookup*. We use the inverse suffix array and this extended version of *substring* in Section 2.9.

2.8.2 High-Order Entropy-Compressed Suffix Arrays

We consider the task of attaining entropy bounds for the usage of space in the CSA by using our unified algorithmic framework for Φ_k at each level k, which contributes the bulk of the space that the CSA uses. In the rest of this section, we prove the tradeoffs shown in Table 2.1 for the space and time complexity of a CSA and its supported operations as given in Definition 2.

Theorem 9 (Time-Efficient Entropy-Compressed Suffix Arrays). *Implementing a* CSA *uses* $nH_h \lg\lg_\sigma n + O\big(n(\lg\lg\lg_\sigma n/\lg\lg_\sigma n + \lg\lg n/\lg_\sigma n + \lg\sigma/\lg_\sigma n) + \sigma^h(n^\beta + \sigma)\big)$ *bits and* $O\big(n\lg\sigma + \sigma^h(n^\beta+\sigma)\big)$ *preprocessing time for compress, for any arbitrarily small constant* $0 < \beta < 1$*. (The space increases to* $O(n) = o(n\lg\sigma)$ *when* σ *is non-constant.) Each lookup takes* $O(\lg\lg_\sigma n)$ *time and each substring call for c symbols takes the cost of lookup plus* $O(c/\lg_\sigma n)$ *time.*

It is worth noting that the space bound in Theorem 9 is $nH_h \lg\lg_\sigma n + o(n\lg\sigma)$ bits when $h+1 \le \alpha\lg_\sigma n$ for any arbitrary positive constant $\alpha < 1$. (We fix β such that $\alpha+\beta < 1$.) [12] When $\lg\sigma = \Theta(\lg n)$, the space bound reduces to $O(nH_h) + o(n\lg\sigma)$ bits and *lookup* time is $O(1)$. A better space usage can be obtained with the following tradeoff.

Theorem 10 (Space-Efficient Entropy-Compressed Suffix Arrays). *Implementing a* CSA *uses* $\epsilon^{-1}nH_h + O\big(n\lg\lg n/\lg_\sigma^\epsilon n + \sigma^h(n^\beta+\sigma)\big)$ *bits and* $O\big(n\lg\sigma + \sigma^h(n^\beta+\sigma)\big)$ *preprocessing time for compress, for any arbitrarily small constants* $0 < \beta < 1$ *and* $0 < \epsilon \le 1/2$*. Each lookup takes* $O\big((\lg_\sigma n)^{\epsilon/1-\epsilon}\lg\sigma\big)$ *time and each substring call for c symbols takes the cost of lookup plus* $O(c/\lg_\sigma n)$ *time.*

For an alphabet of non-constant size, we can use the following corollary of Theorem 10:

Corollary 3 (Space-Efficient Entropy-Compressed Suffix Arrays). *Implementing a* CSA *uses* $\epsilon^{-1}nH_h + O\big(n + \sigma^h(n^\beta+\sigma)\big)$ *bits and* $O\big(n\lg\sigma + \sigma^h(n^\beta+\sigma)\big)$

[12]The assumption on $h+1 \le \alpha\lg_\sigma n$ is reasonable since Luczak and Szpankowski [LS97] show that the average phrase length of the Lempel-Ziv encoding for ergodic sources is $O(\lg n)$ bits.

preprocessing time for compress, for any arbitrarily small constants $0 < \beta < 1$ *and* $0 < \epsilon \leq 1/2$*. Each lookup takes* $O\big((\lg_\sigma n)^{\epsilon/1-\epsilon}\big)$ *time and each substring call for c symbols takes the cost of lookup plus* $O(c/\lg_\sigma n)$ *time.*

The space bound in Theorem 10 and Corollary 3 is $\epsilon^{-1} nH_h + o(n \lg \sigma)$ when $h+1 \leq \alpha \lg_\sigma n$ for $\sigma = \omega(1)$ and any arbitrary positive constant $\alpha < 1$ (we fix β such that $\alpha + \beta < 1$). A special case gives the best space bound in this chapter:

Theorem 11 (Nearly Space-Optimal Entropy-Compressed Suffix Arrays). *Implementing a* CSA *uses* $nH_h + O\big(n \lg\lg n/\lg_\sigma n + \sigma^h(n^\beta + \sigma)\big)$ *bits and* $O(n \lg \sigma + \sigma^h(n^\beta + \sigma))$ *preprocessing time for compress, for any arbitrarily small constant* $0 < \beta < 1$*. Each lookup takes* $O(\lg^2 n/\lg\lg n)$ *time and each substring call for c symbols takes the cost of lookup plus* $O(c \lg \sigma)$ *time.*

The CSA in Theorem 11 is a nearly space-optimal self-index in that it uses the same space as the *compressed* text—nH_h bits—plus the lower-order terms for the text indexing directories. For example, we get $nH_h + O(n \lg\lg n/\lg n)$ bits when $\sigma = O(1)$ and $h + 1 \leq \alpha \lg_\sigma n$ for any arbitrary constant $\alpha < 1$ (we fix β such that $\alpha + \beta < 1$). All space bounds mentioned above include implicitly the cost $M(T, \Sigma, h)$ of the statistical model, which is dominated by the other lower-order terms.

Compressed Representation of the Neighbor Function Φ_k

We now show how to obtain entropy bounds for implementing Φ_k at each level k of a CSA. We refer to the machinery discussed for the implementation of Φ in Section 2.7.2. Since $\Phi = \Phi_0$ and $n = n_0$, we can use either of Theorems 7 and 8 for level $k = 0$. Hence we restrict our focus on level $k > 0$, for which we are interested in extending the bounds of Theorem 8. We introduce some useful notation to this end. We denote the number of elements at level k by $n_k = n/2^k$, and the number of elements at level k that are in context x by n_k^x. Similarly, we 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 $\langle x, y\rangle$. Note that $n_k = \sum_x n_k^x = \sum_y n_k^y = \sum_{x,y} n_k^{x,y}$.

Lemma 24. *For any level k, the Φ_k function can be represented in a compressed format using $nH_h + O(n_k + \sigma^{2^k+h})$ bits of space, so that each call to Φ_k takes $O(1)$ time.*

Proof. We conceptually partition the symbols of the text T into $n/2^k$ non-overlapping segments of 2^k symbols each, assuming without loss of generality that n is a multiple of 2^k. We refer to each segment as a "meta-symbol" and we can regard the text T as a new text T_k consisting of $n/2^k$ meta-symbols over the alphabet $\Sigma' = \Sigma^k$. (These meta-symbols are precisely those corresponding to the Σ' lists at level k. We still draw contexts of length h from the original text T.) Note that SA_k is the suffix array for T_k and Φ_k is the corresponding Φ function at level k. Consequently, we can implement Φ_k along the lines described in Section 2.7.2. However, a direct application of Theorem 8 to T_k for the analysis of the space usage requires some observations to obtain the claimed bounds.

First, we need to refine the analysis by reviewing the space complexity of the auxiliary data structures in Section 2.7.2, indexing them by k to denote their use at level k. Directories G_k and F_k require $O(n_k)$ bits of space by Lemma 21 (where $l = l_k, n = n_k$), using the fact that $\lg \binom{a}{b} \leq a$. Directories L_k^y, for all lists y at level k, occupy a total of $O(n_k + \sigma^{2^k+h})$ bits by Lemma 22 and Lemma 23 (where $n = n_k$ and $s \leq n_k$ is an upper bound on the number of sublists at level k).

Second, we need to relate the high-order entropy of T_k with H_h in our analysis. The current T_k is built on all of the *even* text positions of T_{k-1}. Similarly, there is also text built on *odd* positions. Let $T_k^e = T_k$ and T_k^o denote the two different ways of merging every two symbols of T_{k-1}. When reflected to T, note that T_k^o and T_k^e overlap in T except for $O(2^k)$ initial or final symbols in T. Hence, they essentially encode the same information. We bound the entropy of T_k^o and T_k^e together, showing that their total entropy is no more than $nH'_h + nH'_{h+1}$ bits, which can be bounded by $2nH_h$ by Theorem 1. Hence, representing any of the two requires at most $nH_h + O(2^k \lg \sigma)$ bits, proving the lemma (since we need that bound for T_k^e). For the sake of clarity, let $n_o^{x,yz}$ denote the number of occurrences in T_k^o of the concatenated sequence yzx, where $y, z \in \sigma^{2^k}$ and $x \in P_h^*$. We set $n_o^{x,yz} = 0$ when x is not aligned to a position of T_k^o reflected in T. We similarly define $n_e^{x,yz}$ for T_k^e. Then, their entropy is

$$nH'_h(T_k^o) + nH'_h(T_k^e) = \\ \sum_{x \in P_h^*} \lg \binom{n_o^x}{n_o^{x,11}, n_o^{x,12}, \ldots, n_o^{x,\sigma^{2^k}\sigma^{2^k}}} + \sum_{x \in P_h^*} \lg \binom{n_e^x}{n_e^{x,11}, n_e^{x,12}, \ldots, n_e^{x,\sigma^{2^k}\sigma^{2^k}}} \tag{2.21}$$

Using Equation (2.14), we separate the terms in (2.21) fully into a product of binomial coefficients with $\sigma^{2^{k+1}}$ total terms. Then, since $\binom{a}{b}\binom{c}{d} \le \binom{a+c}{b+d}$ for all positive $a \ge b, c \ge d$, we simplify by combining the respective terms in (2.21) to get

$$\begin{aligned}
\sum_{x \in P_h^*} \lg \binom{n^x}{n^{x,11}, n^{x,12}, \ldots, n^{x,\sigma^{2^k}\sigma^{2^k}}} &= \sum_{x \in P_h^*} \lg \left(\frac{n^x!}{\prod_{y,z \in \sigma^{2^k}} n^{x,yz}!} \right) \\
&= \sum_{x \in P_h^*} \lg \left(\frac{n^x!}{\prod_{y,z \in \sigma^{2^k}} n^{x,yz}!} \right) \left(\frac{\prod_{z \in \sigma^{2^k}} n^{x,z}!}{\prod_{z \in \sigma^{2^k}} n^{x,z}!} \right) \\
&= \sum_{x \in P_h^*} \lg \left(\frac{n^x!}{\prod_{z \in \sigma^{2^k}} n^{x,z}!} \right) \left(\frac{\prod_{z \in \sigma^{2^k}} n^{zx}!}{\prod_{y,z \in \sigma^{2^k}} n^{zx,y}!} \right) \\
&= nH'_h + nH'_{h+1}
\end{aligned}$$

by the definition of high-order empirical entropy H'_h from equation (2.9) and multinomial coefficients. Thus, one of the two texts at level k requires at most nH_h bits to encode (since $nH'_h \le nH_h$ and $nH'_{h+1} \le nH'_h$). We build level k on this text, storing one bit to indicate whether we are storing even or odd text positions at each level, thus proving the lemma. $\square$

Bounds for the Entropy-Compressed Suffix Array

We have almost all of the pieces we need to prove Theorems 9–11 for CSA. We begin with the proof of Theorem 9. We define $\ell = \lg\lg_\sigma n$ to be the last level in the CSA, as given in Section 2.8.1. We introduce a special level $\ell' = \ell - O(1)$, such that $\sigma^{2^{\ell'}} = O(n^\beta)$ for any arbitrary constant $0 < \beta < 1$. Our choice of ℓ' implies that $2^{\ell'} = \Theta(\lg_\sigma n)$ and $2^{\ell-\ell'} = O(1)$.

Instead of storing all levels as discussed in Section 2.8.1, we only store the levels $k = 0, \lg\ell', \lg\ell'+1, \lg\ell'+2, \ldots, \ell'-1, \ell'$ of the recursion in the CSA. (Notice the gap between 0 and $\lg\ell'$, and the gap between ℓ' and ℓ.) For each of these levels up to ℓ', we store a bitvector B_k and a neighbor function Φ_k as described in Section 2.8.1, with their space detailed in the points below:

1. Bitvector B_0 stores $n_{\lg\ell'}$ entries out of a universe of size n, implemented as an indexable dictionary [RRR02] using $O(n_{\lg\ell'}\lg(n/n_{\lg\ell'})) = O(n\lg\lg\lg_\sigma n/\lg\lg_\sigma n)$ bits. For $\lg\ell' \le k \le \ell'-1$, bitvector B_k stores $n_k/2$ entries out of a universe of size n_k, implemented as an indexable dictionary requiring $O(n_k)$ bits. Hence, the total contribution is $O(n\lg\lg\lg_\sigma n/\lg\lg_\sigma n)$ bits.
2. Neighbor function Φ_k is implemented as described in Section 2.8.2. The space bounds are stated in Theorems 7–8 when $k=0$, either $nH_h + O(n\lg\lg n/\lg_\sigma n) + g'_h\lg(1+n/g'_h)$ or $nH_h + O(n) + g'_h\lg(1+n/g'_h)$ bits of space, where $g'_h = O(\sigma^{h+1})$. For $k > 0$, we use Lemma 24, which gives $\sum_{k=\lg\ell'}^{\ell'}(nH_h + O(n_k + \sigma^{2^k+h})) < nH_h(\lg\lg_\sigma n - 1) + O(n_{\lg\ell'} + \sigma^{2^{\ell'}+h})$ bits, where the second term can be bounded as $O(n_{\lg\ell'} + \sigma^{2^{\ell'}+h}) = O(n/\lg\lg_\sigma n + \sigma^h n^\beta)$.
3. Level $k = \ell$ should explicitly store the suffix array SA_ℓ and the inverted suffix array SA_ℓ^{-1}, according to what we described in Section 2.8.1. To significantly reduce the space usage, we now store the arrays at level $\ell + \lg t(n)$ where we fix $t(n) = \lg\lg_\sigma n$. Hence we store $SA_{\ell+\lg t(n)}$, $SA^{-1}_{\ell+\lg t(n)}$, along with an array

$LCP_{\ell+\lg t(n)}$ for the longest common prefix information [MM93] to allow fast searching in $SA_{\ell+\lg t(n)}$, with a total space contribution of $O(n \lg \sigma / \lg \lg_\sigma n)$ bits for level $\ell + \lg t(n)$.

Summing up the bounds in points 1–3, we obtain a final bound of $nH_h \lg \lg_\sigma n + O\big(n\big(\lg \lg \lg_\sigma n / \lg \lg_\sigma n + \lg \lg n / \lg_\sigma n + \lg \sigma / \lg_\sigma n\big) + \sigma^h(n^\beta + \sigma)\big)$ bits of space required for the CSA, for any arbitrarily small constant $0 < \beta < 1$. Note that the latter bound is $nH_h \lg \lg_\sigma n + o(n \lg \sigma) + O(\sigma^h(n^\beta + \sigma))$. The space has an additional term $O(n) = o(n \lg \sigma)$ when σ is non-constant, since we use Theorem 8 for level $k = 0$.

Building the above data structures is a variation of what was done in [GV05]; thus it takes $O\big(n \lg \sigma + \sigma^h(n^\beta + \sigma)\big)$ time to *compress* (as given in Definition 2). The *lookup* operation requires $O(2^{\lg \ell'} + \ell' + 2^{\ell + \lg t(n) - \ell'}) = O(\lg \lg_\sigma n)$ time because accessing any of the data structures in any level requires constant time. (Note that, for level $k = 0$, we use Theorem 7 if $\sigma = O(1)$ or Theorem 8 otherwise). A *substring* query for c symbols requires $O(c / \lg_\sigma n + \lg \lg_\sigma n)$ time since $\Phi_{\ell'}$ decompresses $2^{\ell'} = \Theta(\lg_\sigma n)$ symbols at a time, as we remarked in Section 2.8.1. This completes the proof of Theorem 9.

We now discuss the complexity of CSA that leads to Theorem 10. We keep a constant number $1/\epsilon$ of the levels as in [GV05], where $0 < \epsilon \leq 1/2$. In particular, we store level 0, level ℓ', and then one level every other $\lambda \ell'$ levels; in sum, $1 + 1/\lambda = 1/\epsilon$ levels, where $\lambda = \epsilon/(1-\epsilon)$ with $0 < \lambda < 1$. Each such level $k \leq \ell'$ stores the following data structures:

- A directory D_k (in place of B_k in point 1 above) storing the $n_{k+\lambda\ell'}$ (or n_ℓ when $k = \ell'$) entries of the next sampled level. Note that D_0, which stores $n_{\lambda\ell'}$ entries out of a universe of size n, requires just $O(n_{\lambda\ell'}\ell') = O(n \lg \lg_\sigma n / \lg_\sigma^\lambda n)$ bits by using an indexable dictionary [RRR02]. Each of the other D_k's add a geometrically decreasing contribution upper bounded by the cost of D_0.

- A neighbor function Φ_k implemented as given in point 2 above. For all levels $k = \lambda\ell', 2\lambda\ell', \ldots$, neighbor function Φ_k contributes a (geometrically decreasing) total of $O(n_{\lambda\ell'}) = O(n/\lg_\sigma^\lambda n)$ bits, in addition to the term of $O(\sigma^{2^{\ell'}+h}) = O(\sigma^h n^\beta)$ as before. Note that the analysis for Φ_0 is as given in point 2 above. The total required space is therefore (where $\lambda > \epsilon$)

$$\epsilon^{-1} n H_h + O\left(\frac{n \lg\lg_\sigma n}{\lg_\sigma^\lambda n} + \frac{n \lg\lg n}{\lg_\sigma n} + \sigma^h n^\beta\right) = \epsilon^{-1} n H_h + O\left(\frac{n \lg\lg n}{\lg_\sigma^\epsilon n} + \sigma^h n^\beta\right). \tag{2.22}$$

- The arrays mentioned in point 3 above, except that we now fix $t(n) = \lg_\sigma^\lambda n \lg \sigma$. Thus, we obtain a total space contribution of $O(n \lg \sigma / t(n)) = O(n/\lg_\sigma^\lambda n)$ bits.

In sum, we obtain a total space complexity that is bounded by Equation (2.22). Thus, we are able to save space at a small cost to *lookup*, namely, $O(2^{\lambda\ell'} \lg \sigma + (1/\epsilon - 1)2^{\lambda\ell'} + 2^{\ell + \lg t(n) - \ell'})$ time, where the $\lg \sigma$ factor in the first term is due to the implementation of Φ_0 with the bounds of Theorem 7. Simplifying, we obtain $O(\lg_\sigma^\lambda n \lg \sigma + t(n)) = O(\lg_\sigma^\lambda n \lg \sigma) = O((\lg_\sigma n)^{\epsilon/1-\epsilon} \lg \sigma)$. The *substring* operation for c symbols requires an additional $O(c/\lg_\sigma n)$ time. We can drop the $\lg \sigma$ factor to $O(1)$ in Corollary 3 by using Theorem 8 for the analysis of Φ_0. Building the above data structures is again a variation of what was done for Theorem 9, so *compress* requires $O(n \lg \sigma + \sigma^h n^\beta)$ time, thus proving Theorem 10.

Finally, we prove Theorem 11, which is an interesting special case by a simple modification of the scheme described above. Here we just keep levels 0 and $\ell + \lg t(n)$ where $t(n) = \lg n / \lg\lg n$. We store the following data structures:

- Dictionary D_0 stores $n_{\ell + \lg t(n)}$ entries over a universe of size n in $O(n_{\ell+\lg t(n)}(\ell + \lg t(n))) = O(n(\lg\lg_\sigma n + \lg t(n))/(t(n) \lg_\sigma n))$ bits using an indexable dictionary [RRR02].
- The neighbor function Φ_0 from point 2 above, with the bounds of Theorem 7.
- The three arrays as given in point 3 above, using $O(n \lg \sigma / t(n))$ bits.

Thus, the total space is $nH_h + O(n(\lg\lg_\sigma n + \lg t(n))/(t(n)\lg_\sigma n) + n\lg\lg n/\lg_\sigma n + n\lg\sigma/t(n)) = nH_h + O(n\lg\lg n/\lg_\sigma n)$ bits. We also have to add $O\big(\sigma^{h+1}\lg(1 + n/\sigma^{h+1})\big)$ bits for the statistical model. The *lookup* cost is bounded by $O(2^{\ell+\lg t(n)}\lg\sigma) = O(t(n)\lg_\sigma n\lg\sigma) = O(\lg^2 n/\lg\lg n)$, where the $\lg\sigma$ factor comes from the cost of a call to Φ_0 (with the bounds of Theorem 7). Similarly, decompressing each symbol in *substring* has a $O(\lg\sigma)$ cost.

2.9 Applications to Text Indexing

We use the CSA as an integral component of an efficient text indexing structure that attains the hth-order entropy for a text T of n symbols over alphabet Σ. *Throughout this section, we assume that $h + 1 \leq \alpha\lg_\sigma n$ for any arbitrary constant $\alpha < 1$ to guarantee that the encoding of the empirical statistical model requires $o(n)$ bits.*[13] Our high-order entropy-compressed text indexes support fast searching of a pattern P of length m in $O(m + \text{polylg}(n))$ time with only $nH_h + o(n)$ bits, where nH_h is the information-theoretic upper bound on the number of bits required to encode the text T of length n (cf. Section 2.2). We also describe a text index that takes $o(m)$ search time *and* uses $o(n)$ bits on highly compressible texts with a small-sized alphabet Σ. The full list of tradeoffs for the space and time complexity of compressed text indexing is shown in Table 2.2.

2.9.1 High-Order Entropy-Compressed Text Indexing

We now present our search of a pattern P of length m in the CSA for T. We need the following pattern matching tool to search for P in a sequence of contiguous suffixes

[13]This condition is not satisfied if keeping the suffix array *uncompressed* for the text T requires nearly the same space as encoding the hth-order empirical statistics of T. Hence T is not a low-entropy text.

stored in the CSA, in compressed form, where the proof of Lemma 25 is given in Section 2.9.2.

Lemma 25 (Pattern Matching Tool). *Given a sequence of r consecutive suffixes stored in the* CSA, *we can search for the leftmost and the rightmost of these suffixes having a pattern P of length m as a prefix, in $O(m + r)$ symbol comparisons plus $O(r)$ lookup and substring operations.*

We show how to search P using the CSA and the tool in Lemma 25. We first perform a binary 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 [MM93]. (The term $t(n)$ depends on the implementation of the CSA as described in Section 2.8.2.) Because we have the longest common prefix information, the binary search requires only $O(m)$ symbol comparisons plus $O(\lg n)$ *lookup* and *substring* operations. At that point, we locate $r = 2^{\ell+\lg t(n)} = O(t(n) \lg_\sigma n)$ contiguous suffixes stored, in compressed form, in the CSA. We run the pattern matching tool in Lemma 25 on these r suffixes, at the cost of $O(m + t(n) \lg_\sigma n)$ symbol comparisons and $O(t(n) \lg_\sigma n)$ calls to *lookup* and *substring*, which is also the asymptotic cost of the whole search. The following results show several tradeoffs that we obtain with the simple search scheme described so far.

Theorem 12. *Given a text T of n symbols over an alphabet Σ, we can replace T by a* CSA *occupying $\epsilon^{-1} n H_h + O(n \lg \lg n / \lg_\sigma^\epsilon n)$ bits, so that searching for 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 $0 < \epsilon \leq 1/2$. Reporting each occurrence of the pattern P will take no more than $O((\lg n)^{(1+\epsilon)/(1-\epsilon)} (\lg \sigma)^{(1-3\epsilon)/(1-\epsilon)})$ time.*

Proof. Using Theorem 10, we have $t(n) = \lg_\sigma^\lambda n \lg \sigma$, where $\lambda = \epsilon/(1-\epsilon)$. The $O(m + t(n) \lg_\sigma n)$ symbol comparisons give a contribution of $O((m + \lg_\sigma^{1+\lambda} n \lg \sigma)/\lg_\sigma n) =$

$O(m/\lg_\sigma n + \lg_\sigma^\lambda n \lg \sigma)$, since we can decompress and compare $\Theta(\lg_\sigma n)$ adjacent symbols with $O(1)$ RAM operations. The $O(t(n)\lg_\sigma n) = O(\lg_\sigma^{1+\lambda} n \lg \sigma)$ calls to *lookup* and *substring* (see Lemma 25) give a contribution of $O(\lg_\sigma^{1+2\lambda} n \lg^2 \sigma) = O((\lg n)^{(1+\epsilon)/(1-\epsilon)} (\lg \sigma)^{(1-3\epsilon)/(1-\epsilon)})$. □

For example, fixing $\epsilon = 1/2$ in Theorem 12 when $\sigma = O(1)$, we obtain a search time of $O(m/\lg n + occ \times \lg^3 n)$ with a self-index occupying $2nH_h + O(n \lg\lg n/\sqrt{\lg n})$ bits, where *occ* is the number of occurrences reported. We can reduce the space to nH_h bits plus a lower-order term, obtaining the first nearly space-optimal self-index with polylg(n) reporting time.

Theorem 13. *Given a text of n symbols over an alphabet Σ, we can replace it by a* CSA *occupying nearly optimal space, i.e., $nH_h + O(n \lg\lg n/\lg_\sigma n)$ bits, so that searching for a pattern of length m takes $O(m \lg \sigma + \lg^4 n/(\lg^2 \lg n \lg \sigma))$ time. Reporting each pattern occurrence takes $O(m \lg \sigma + \lg^4 n/(\lg^2 \lg n \lg \sigma))$ time.*

Proof. Using Theorem 11, we have $t(n) = \lg n/\lg\lg n$. The $O(m + t(n)\lg_\sigma n)$ symbol comparisons contribute $O(m \lg \sigma + \lg^2 n/\lg\lg n)$ time in total, while the $O(t(n)\lg_\sigma n) = O(\lg^2 n/(\lg\lg n \lg \sigma))$ calls to *lookup* and *substring* contribute $O(\lg^4 n/(\lg^2 \lg n \lg \sigma))$. □

If we augment the CSA to obtain the hybrid multi-level data structure in [GV05], we can improve the lower-order terms in the search time of Theorem 12, where $t(n) = \lg_\sigma^\lambda n \lg \sigma$ and $\lambda = \epsilon/(1-\epsilon) > \epsilon$. We use a sparse suffix tree storing every other $(t(n)\lg n)$th suffix using $O(n/t(n)) = O(n/\lg_\sigma^\epsilon n)$ bits to locate a portion of the (compressed) suffix array storing $O(t(n)\lg n)$ suffixes. However, we do not immediately run our pattern matching tool from Lemma 25; instead, we employ a nested sequence of space-efficient Patricia tries [MRS01a] of size $\lg^{\omega-\lambda} n$ until we are left with segments of $r = \lg_\sigma^\lambda n$ adjacent suffixes in the CSA, for any fixed value of

$1 > \omega \geq 2\lambda > 0$. This scheme adds $O(n/r) = O(n/\lg_\sigma^\epsilon n)$ bits to the self-index, allowing us to restrict the search of pattern P to a segment of r consecutive suffixes in the CSA. At this point, we run our pattern matching tool from Lemma 25 on these r suffixes to identify the leftmost occurrence of the pattern.

Theorem 14. *Given a text of n symbols over an alphabet Σ, we can replace it by a hybrid* CSA *occupying $\epsilon^{-1} n H_h + O(n \lg\lg n / \lg_\sigma^\epsilon n)$ bits, so that searching for a pattern of length m takes $O(m/\lg_\sigma n + \lg^\omega n \lg^{1-\epsilon} \sigma)$ time, for any fixed value of $1 > \omega \geq 2\epsilon/(1-\epsilon) > 0$ and $0 < \epsilon \leq 1/3$.*

Proof. Searching in the sparse suffix tree takes $O(m/\lg_\sigma n + \lg_\sigma^\lambda n \lg \sigma)$ time as in [GV05], where the second term is our *lookup* cost in Theorem 10 with $\lambda = \epsilon/(1-\epsilon)$. Then, the search goes through a constant number of space-efficient Patricia tries with $O(\lg^{\omega-\lambda} n)$ calls to *lookup* and *substring*, each of $O(\lg_\sigma^\lambda n \lg \sigma)$ time, requiring a total of $O(\lg^\omega n \lg^{1-\epsilon} \sigma)$ time by Theorem 10. Finally, the pattern matching tool is run on a segment of $r = O(\lg_\sigma^\lambda n)$ suffixes, in $O(\lg_\sigma^{2\lambda} n \lg \sigma) = O(\lg^\omega n \lg^{1-\epsilon} \sigma)$ time. The cost of comparing $\Theta(\lg_\sigma n)$ symbols at a time and decompressing them sums to $O(m/\lg_\sigma n)$, where the additional cost of *substring* is accounted for above. $\square$

For low-entropy texts, we provide the first self-index with small alphabets that is sublinear both in space and in search time.

Corollary 4. *When $H_h = o(1)$ for a text over an alphabet of size $\sigma = O(1)$, the self-index in Theorem 14 occupies just $o(n)$ bits and requires $o(m)$ search time. Reporting each occurrence takes $o(\lg n)$ time.*

2.9.2 A Pattern Matching Tool

In this section, we prove Lemma 25 by describing the implementation of the following pattern matching tool. Given a list of r sequences $S_1 \leq \cdots \leq S_r$ in lexicographical order, the pattern matching tool identifies the least sequence S_i having P as a prefix in $O(m + r)$ time. (Identifying the greatest such sequence is analogous.) We first assume that these r suffixes are explicitly given. Next, we show how to adapt the tool when these suffixes are stored, in compressed form, in the CSA.

Our search tool is reminiscent of the Patricia search [Mor68], the Hirschberg's sequential search [Hir78], and the Bit-Tree search [Fer92], as we only need one full comparison of P against a suffix. Our tool examines the sequences $S_1, \ldots, S_r$ in left-to-right order. We start out by comparing the symbols of P against the symbols of S_1 consecutively until there is a mismatch. We then find the first match in S_2 starting with the symbol that caused the mismatch with S_1. We repeat this process starting at S_2. We stop when we have examined all the sequences unsuccessfully (declaring that there is no occurrence of P), or we succeed in matching the symbols of P at sequence S_i. 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. Increment k until either $k > m$ or $S_i[k] \neq P[k]$. If $k > m$, go to step 4; otherwise, find the smallest $j > i$ such that $S_j[k] = P[k]$.
3. If such j does not exist, declare that P is not the prefix of any sequence and quit with a failure. Otherwise, assign the value of j to i.
4. If $k \leq m$, 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; declare a failure otherwise.

Denoting the positions assigned to i in step 3 with $i_1 < i_2 < \cdots < i_k$, we observe that we do not access the first $k - 1$ symbols of $S_{i_{k-1}+1}, \ldots, S_{i_k}$, 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. $k > m$) do we set $i = i_m$ and perform a full comparison of P

against S_i in order to determine if there is really a match. This results in a correct method notwithstanding potential mismatches.

Lemma 26. *Given a list of r sequences $S_1, \ldots, S_r$ in lexicographical order, let S_i be the sequence identified by our search tool. If the pattern 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.*

Proof. Suppose P is a prefix of S_i, where S_i was identified by our search tool. We first show that P is not a prefix of $S_1, \ldots, S_{i-1}$. Suppose by contradiction that a sequence S_f has P as a prefix, where $f < i$. Suppose that we are matching the kth symbol of P at the time we examine S_f. Since P is a prefix of S_f, we have a match and our search tool scans the $(k+1)$st symbol in P, the $(k+2)$nd symbol in P and so on, matching all of them with S_f. Hence, our search tool identifies S_f with $f \neq i$, giving a contradiction. This logic proves the first part of the lemma; namely that S_i is the least sequence having P as a prefix, because we consider the sequences S_i in lexicographical order.

To prove the second part, we know that our search tool fails to match P. To see why no sequence in $S_1, \ldots, S_r$ has P as a prefix, note that $S_1, \ldots, S_{i-1}$ cannot have P as a prefix as shown in the previous paragraph. We also have to show this property for the remaining sequences $S_i, \ldots, S_r$. Suppose by contradiction that a sequence S_j, with $j \geq i$, has P as a prefix. Let k be the position of the rightmost symbol in P that we compare to S_j. Our method implies that the kth symbol in S_j is different from that of P. Hence, P cannot be a prefix of S_j, giving the contradiction.

Finally, the time required is $O(m + r)$, as each comparison in our method contributes to at most $2m$ matches and at most r mismatches. $\square$

We now evaluate how the time complexity is affected if $S_1, \ldots, S_r$ are implicitly stored in the CSA, say, at consecutive positions $q + 1, \ldots, q + r$ for a suitable value of q. To use our search tool, we need to decompress starting from the kth symbol of a suffix S_i by knowing its position $q+i$ in the CSA. (Recall that $SA[q+i]$ contains the starting position of S_i in the text.) To this end, it suffices to decompress the *first* symbols in the suffix at position $SA^{-1}\big[SA[q+i]+k-1\big]$ in the CSA, where SA and SA^{-1} denote the suffix array and its inverse (as mentioned in Definition 2). Equivalently, the latter suffix S_j can be obtained by removing the first $k-1$ symbols from S_i since $j = SA[q+i]+k-1$. This scheme only requires a constant number of *lookup* operations and a single *substring* operation, with a cost that is independent of the value of k, thus proving Lemma 25.

2.10 Conclusions

We have presented a unified algorithmic framework for analysis of compression and text indexing. We described two techniques—a *context-sensitive partitioning scheme* and the *wavelet tree*—to provide the first optimal space bounds for the Burrows-Wheeler transform aside from lower-order terms. We then used this critical framework to develop a text indexing structure based on a high-order entropy-compressed suffix array that exhibit several tradeoffs between occupied space, search, and decompression time. We described how to implement them 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 $O(m \lg \sigma + \text{polylg}(n))$ time. Our scheme provides the first self-index that asymptotically realizes the high-order entropy H_h per symbol of the text. We also proved how to achieve the first self-index with sublinear size $o(n)$ in bits and sublinear query time $o(m)$ for low-entropy texts over an alphabet of constant size.

The most immediate goal is to address whether a compressed full-text index with $nH_h + O(\text{polylg}(n))$ bits and $O(m + \text{polylg}(n))$ query time exists. If not, it would separate indexing from compression for very low-entropy strings. Beyond that, we would like to achieve $nH_h + O(n \lg\lg n / \lg_\sigma n)$ bits with an optimal $O(m/\lg_\sigma n + occ)$ search time. A compelling problem is to improve the time for *lookup* so that each call takes constant time. Another interesting challenge would be to support approximate matches (those that match patterns with some threshold of error).

Original	*Sorted*			*Mappings*			*Suffix Array*	
Q	F		L	i	$LF(i)$	$\Phi(i)$	$SA[i]$	
mississippi#	i	ppi#missis	s	1	8	7	8	ippi#
#mississippi	i	ssippi#mis	s	2	9	10	5	issippi#
i#mississipp	i	ssissippi#	m	3	5	11	2	ississippi#
pi#mississip	i	#mississip	p	4	6	12	11	i#
ppi#mississi	m	ississippi	#	5	12	3	1	mississippi#
ippi#mississ	p	i#mississi	p	6	7	4	10	pi#
sippi#missis	p	pi#mississ	i	7	1	6	9	ppi#
ssippi#missi	s	ippi#missi	s	8	10	1	7	sippi#
issippi#miss	s	issippi#mi	s	9	11	2	4	sissippi#
sissippi#mis	s	sippi#miss	i	10	2	8	6	ssippi#
ssissippi#mi	s	sissippi#m	i	11	3	9	3	ssissippi#
ississippi#m	#	mississipp	i	12	4	5	12	#

Table 2.3: Matrix Q for the BWT containing the cyclic shifts of text T = mississippi# (column '*Original*'). Sorting of the rows of Q, in which the first (F) and last (L) symbols in each row are separated (column '*Sorted*'). Functions LF and Φ for each row of the sorted Q (column '*Mappings*'). Suffix array SA for T (column '*Suffix Array*').

context x	**list** `i`	**list** `m`	**list** `p`	**list** `s`	**list** `#`
`i`	$\emptyset$	$\langle 3\rangle$	$\langle 4\rangle$	$\langle 1,2\rangle$	$\emptyset$
`m`	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\langle 5\rangle$
`p`	$\langle 7\rangle$	$\emptyset$	$\langle 6\rangle$	$\emptyset$	$\emptyset$
`s`	$\langle 10,11\rangle$	$\emptyset$	$\emptyset$	$\langle 8,9\rangle$	$\emptyset$
`#`	$\langle 12\rangle$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Table 2.4: An example of our conceptual table $\mathcal{T}$, where each sublist $\langle x, y\rangle$ contain $n^{x,y}$ entries. The contexts x are associated with rows and the lists y are associated with columns.

context x	n^x	$\#x$	**list** `i`	**list** `m`	**list** `p`	**list** `s`	**list** `#`
`i`	4	0	$\emptyset$	$\langle 3\rangle$	$\langle 4\rangle$	$\langle 1,2\rangle$	$\emptyset$
`m`	1	4	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\langle 1\rangle$
`p`	2	5	$\langle 2\rangle$	$\emptyset$	$\langle 1\rangle$	$\emptyset$	$\emptyset$
`s`	4	7	$\langle 3,4\rangle$	$\emptyset$	$\emptyset$	$\langle 1,2\rangle$	$\emptyset$
`#`	1	11	$\langle 1\rangle$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Table 2.5: The sublists of Table 2.4 in normalized form. The value of n^x is defined as in Equation (2.9) and indicates the interval length in the row for context x. The value $\#x$ should be added to the sublists' entries in row x to obtain the same entries in Table 2.4.

Chapter 3

When Indexing Equals Compression: Experiments with Compressing Suffix Arrays and Applications

3.1 Introduction

Suffix arrays and suffix trees are ubiquitous data structures at the heart of several text and string algorithms. They are used in a wide variety of applications, including pattern matching, text and information retrieval, Web searching, and sequence analysis in computational biology [Gus97b]. 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.

The suffix tree is a powerful text index (in the form of a compact trie) whose leaves store each of the n suffixes contained in the text T. Suffix trees [MM93, McC76] allow fast, general searching of patterns in T in $O(m \lg |\Sigma|)$ time, but require roughly $4n \lg n$ bits of space—16 times the size of the text itself, in addition to needing a copy of the text. The suffix array is another well-known index structure. It maintains the permuted order of $1, 2, \ldots, n$ that corresponds to the locations of the suffixes of the text in lexicographically sorted order. Suffix arrays [GBS92, MM93] (that also store the length of the longest common prefix) are nearly as good at searching. Their search time is $O(m + \lg n)$ time, but they require a copy of the text; the space cost is only $n \lg n$ bits (which can be reduced about 40% in some cases).

There are a number of other common indexes that give access to the text, however, none of these can operate without the text itself. Compressed suffix arrays [GV05, Rao02, Sad03, Sad02b] and opportunistic FM-indexes [FM05, FM01] represent modern trends in the design of advanced indexes for full-text searching of documents. They support the functionalities of suffix arrays and suffix trees (which are more powerful than classical inverted files [GBS92]), yet they overcome the aforementioned 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 [BM99, Pag01, RRR02].

A key idea in these new schemes is that of *self-indexing*. 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. Self-indexes can thus replace the text as in standard text compression. However, self-indexes support more functionality than standard text compression. In these cases, the indexing scheme is itself a compression method. We focus on these scenarios, where indexing equals compression.

Grossi and Vitter [GV05] developed the compressed suffix array using $2n \lg |\Sigma|$ bits in the worst case with $o(m)$ searching time. Sadakane [Sad03, Sad02b] extended its functionality to a self-index and related the space bound to the order-0 empirical entropy H_0. Ferragina and Manzini devised the FM-index [FM05, FM01], which is based on the Burrows-Wheeler transform (`bwt`) and is the first to encode the index size with respect to the hth-order empirical entropy H_h of the text, encoding in $(5+\epsilon)nH_h + o(n)$ bits. Grossi, Gupta, and Vitter [GGV03] exploited the higher-order entropy H_h of the text to represent a compressed suffix array in just $nH_h + o(n)$ bits. The index is optimal in space, apart from lower-order terms, achieving asymptotically the empirical entropy of the text (with a multiplicative constant of 1). More results appeared subsequently, and we refer the reader to the survey in [NM06a] for the state of the art.

The above self-indexes are so powerful that the text is implicitly encoded in them and is not needed explicitly. Searching decompresses 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.

3.1.1 Our Results

In this chapter, we provide an experimental study of compressed suffix arrays in order to evaluate their practical impact. In doing so, we exploit the properties and intuition of our earlier result [GGV03] and develop a new design that is driven by experimental analysis for enhanced performance. Briefly, we mention the following new contributions. The work in this chapter was a collaborative effort with Luca Foschini, Roberto Grossi, and Jeffrey Scott Vitter.

Since compressed suffix arrays hinge on succinct dictionaries, we provide a new practical implementation of succinct dictionaries that takes less space than the predicted space based on a worst-case analysis. We then use these dictionaries (organized in a *wavelet tree*), along with run-length encoding (RLE) and γ encoding, to achieve a simplified "encoding" for high-order contexts. This construction shows that Move-to-Front (MTF) [BSTW86], arithmetic, and Huffman encoding are not strictly necessary to achieve high-order compression with the Burrows-Wheeler Transform (`bwt`). Recent work of Ferragina et al. [FGMS05] shows how to find an optimal partition of the `bwt` to attain the same goal; we take a different route and show that the wavelet tree implicitly leads to an optimal partition when using RLE and integer encoding.

We then extend the wavelet tree so that its search can be sped up by fractional cascading and an a-priori distribution on the queries. In addition, we describe an algorithm to construct the wavelet tree in $O(n + \min(n, nH_h) \times \lg |\Sigma|)$ time, introducing the novel concept that indexing/compression time should be related to the compressibility of the data. (Said in another way, highly compressible data should not only be more compact when compressed, but should also require less time to index and compress.) Recently Hon, Sadakane, and Sung have shown how to build the compressed suffix array and FM-index in $O(n \lg \lg |\Sigma|)$ time [Sad03]. One of our main results in this chapter is to give an analysis of our practically-motivated structure and show that it still has competitive theoretical guarantees on space consumption, namely, $2nH_h + o(n)$ bits of space.

We also detail a simplified version of our structure which serves as a powerful compressor for the Burrows-Wheeler Transform (`bwt`). In experiments, we obtain a compression ratio comparable to that of `bzip2`. In addition, we go on to obtain a compressed representation of fully equipped suffix trees (and their associated text) in a total space that is comparable to that of the text alone compressed with `gzip`.

In the rest of the chapter, we use 'bps' to denote the average number of bits needed per text symbol or per dictionary entry. In order to get the compression ratio in terms of a percentage, it suffices to multiply bps by 100/8.

3.1.2 Outline of Chapter

The rest of the chapter is organized as follows. In the next section, we build the critical framework in describing our practical dictionaries, providing both theoretical and practical intuition on our choice. We then describe a simple scheme for fast access to our dictionaries in practice. In Section 3.3, we describe our *wavelet tree* structure, which forms the basis for our compression format `wzip`. In Section 3.4, we describe a practical implementation of compressed suffix arrays [GV05, GGV03], grounded firmly with theoretical analysis. In Section 3.5, we discuss a space-efficient implementation of suffix trees. We conclude in Section 3.6.

3.2 A Simple Yet Powerful Dictionary

As previously mentioned, compressed suffix arrays make crucial use of succinct dictionaries. Thus, we first focus on our implementation of them. We recall that succinct dictionaries are constant-time rank and select data structures occupying tiny space. They store t entries chosen from a bounded universe $[0 \dots n-1]$ in $\lceil \lg \binom{n}{t} \rceil \leq n$ bits, plus additional bits for fast access to the entries. The bound comes from the information-theoretic observation that we need $\lceil \lg \binom{n}{t} \rceil$ bits to enumerate each of the $\binom{n}{t}$ possible subsets of $[0 \dots n-1]$. Equivalently, this is the number of bitvectors B of length n (the universe size) with exactly t $\mathbf{1}$s, such that entry x is stored in the dictionary if and only if $B[x] = \mathbf{1}$. The dictionaries support several operations. The function $rank_{\mathbf{1}}(B, i)$ returns the number of **1**s in B up to (and including) position i. The function $select_{\mathbf{1}}(B, i)$ returns the position of the ith **1** in B. Analogous definitions hold for **0**s. The bit $B[x]$ can be computed as $B[x] = rank_{\mathbf{1}}(B, x) - rank_{\mathbf{1}}(B, x-1)$. In the following, we consider the succinct dictionaries called *fully indexable dictionaries* [RRR02], which support the full repertoire of *rank* and *select* for both **0**s and **1**s in $\lceil \lg \binom{n}{t} \rceil + o(n)$ bits.

Let $p(\mathbf{1}) = t/n$ be the empirical probability of finding a **1** in bitvector B, and $p(\mathbf{0}) = 1 - p(\mathbf{1})$. We define the empirical entropy H_0 as

$$H_0 = -p(\mathbf{0}) \lg p(\mathbf{0}) - p(\mathbf{1}) \lg p(\mathbf{1}).$$

As shown in [GGV03], the empirical entropy H_0 can be approximated by $\frac{1}{n} \lg \binom{n}{t}$. Thus, we can think of succinct dictionaries as 0th-order compressors that can also retrieve any individual bit in constant time. Specifically, the data structuring framework in [GGV03] uses suffix arrays to transform succinct dictionaries into a high-order entropy-compressed text index. As a result, we stress the important consideration of dictionaries in practice, since they contribute fast access to data as well as solid, effective compression. In particular, such dictionaries avoid a complete sequential scan of the data when retrieving portions of it. They also provide the basis for space-efficient representation of trees and graphs [Jac89a, MR99].

3.2.1 Practical Dictionaries

We now explore practical alternatives to dictionaries for use in compressed text indexing data structures. When implementing a dictionary D, there are two main space issues to consider:

- The second-order space term $o(n)$, which is often incurred to improve access time to the data, is non-negligible and can dominate the $\lg \binom{n}{t}$ term.
- The $\lg \binom{n}{t}$ term is not necessarily the best possible in practice. As with strings, we can achieve "entropy" bounds that are better than $\lg \binom{n}{t} \sim nH_0$.

Before describing our practical variant of dictionaries, let's focus on a basic representation problem for the dictionary D seen as a bitvector B_D. Do we always need $\lg \binom{n}{t}$ bits to represent B_D? For instance, if D stores the even numbers in a bounded universe of size n, a simple argument based on the Kolmogorov complexity of B_D implies that we can represent this information with $O(\lg n)$ bits. Similarly, if D stores $n/2$ elements of a contiguous interval of the universe, we can again represent this information with $O(\lg n)$ bits. The $\lg \binom{n}{t}$ term treats these two cases the same a random set of $t = n/2$ integers stored in D; thus, the worst-case bound is $\lg \binom{n}{n/2} \sim n$ bits of space. That is, it is a worst-case measure that does not account for the distribution of the **1**s and **0**s inside B_D, which may allow significant compression (as in the previous examples). In other words, the $\lg \binom{n}{t}$ bound only exploits the *sparsity* of the data we wish to retain.

This observation sparks the realization that many of the bitvectors in common use are probably compressible, even if they represent a minority among all possible bitvectors. Is there then some general method by which we can exploit these patterns? The solution is surprisingly simple and uses elementary notions in data compression [WMB99]. We briefly describe those relevant notions.

Run-length encoding (RLE) represents each subsequence of identical symbols (a run) as the pair (ℓ, s), where ℓ is the number of times that symbol s is repeated. For a binary string, we do not need to encode s, since its value will alternate between **0** and **1**. (We explicitly store the first bit.)

The length ℓ is then encoded in some fashion. One such method is the γ code, which represents the length ℓ in two parts: The first encodes $1 + \lfloor \lg \ell \rfloor$ in unary, followed by the value of $\ell - 2^{\lfloor \lg \ell \rfloor}$ encoded in binary, for a total of $1 + 2\lfloor \lg \ell \rfloor$ bits. For example, the γ codes for $\ell = 1, 2, 3, 4, 5, \ldots$ are $\mathbf{1}, \mathbf{01\,0}, \mathbf{01\,1}, \mathbf{001\,00}, \mathbf{001\,01}, \ldots$, respectively. The δ code requires asymptotically fewer bits by encoding $1 + \lfloor \lg \ell \rfloor$ via the γ code rather than in unary, thus requiring $1 + \lfloor \lg \ell \rfloor + 2\lfloor \lg \lg 2\ell \rfloor$ bits. For example, the δ codes for $\ell = 1, 2, 3, 4, 5, \ldots$ are $\mathbf{1}, \mathbf{010\,0}, \mathbf{010\,1}, \mathbf{011\,00}, \mathbf{011\,01}, \ldots$, respectively. Byte-aligned codes are another simple encoding for positive integers. Let $lb(\ell) = 1 + \lfloor \lg \ell \rfloor$, the minimal number of bits required to represent the positive integer ℓ. A byte-aligned code splits the $lb(\ell)$ bits into groups of 7 bits each, prepending a "continuation" bit as most significant to indicate whether there are more bits of ℓ in the next byte. We refer to [WMB99] for other encodings.

We can represent a conceptual bitvector B_D by a vector of nonnegative "gaps" $G = \{g_1, g_2, \ldots, g_t\}$, where $B_D = \mathbf{0}^{g_1}\mathbf{1}\mathbf{0}^{g_2}\mathbf{1}\ldots\mathbf{0}^{g_t}\mathbf{1}$ and each $g_i \geq 0$. We assume that B_D ends with a $\mathbf{1}$; if not, we can use an extra bit to denote this case and encode the final gap length separately. We also assume that $t \leq n/2$ or else we reverse the role of $\mathbf{0}$ and $\mathbf{1}$. Using gap encoding we cannot require less than

$$E(G) = \sum_{i=1}^{t} lb(g_i + 1) \tag{3.1}$$

to store the gaps corresponding to B_D. We now show that $E(G)$ is closely related to the optimal worst-case encoding of B_D, which takes $\lg \binom{n}{t}$ bits.

Fact 1. *For a conceptual bitvector B_D of known length n, such that B_D ends with a $\mathbf{1}$, its gap encoding G satisfies*

$$E(G) < \lg \binom{n}{t} + 1/2 \lg(t(n-t)/n) + \lg e\, [(1/(12t) + 1/(12(n-t)) - 1/(12n+1)] + \lg \sqrt{2\pi},$$

where $t \leq n/2$ is the number of $\mathbf{1}$s in B_D.

Proof. By convexity, the worst-case optimal cost occurs when the gaps are of equal length, i.e. $g_i + 1 \leq n/t$, giving $E(G) = \sum_{i=1}^{t} lb(g_i+1) \leq t\, lb(n/t) \leq t + t\lg(n/t) \leq (n-t)\lg(n/(n-t)) + t\lg(n/t)$, since $t \leq (n-t)\lg(n/(n-t))$ when $t \leq n/2$. By Stirling's inequality, $\lg \binom{n}{t} > t\lg(n/t) + (n-t)\lg(n/(n-t)) - 1/2\lg(t(n-t)/n) - [(1/(12t) + 1/(12(n-t)) - 1/(12n+1)] \lg e - \lg\sqrt{2\pi}$, thus proving the fact. □

An approach that works better in practice, although not quite as well in the worst case, is to represent B_D by the vector of positive run-length values $L = \{\ell_1, \ell_2, \ldots, \ell_j\}$ (with $j \leq 2t$ and $\sum_i \ell_i = n$) where either $B_D = \mathbf{1}^{\ell_1}\mathbf{0}^{\ell_2}\mathbf{1}^{\ell_3}\ldots$ or $B_D = \mathbf{0}^{\ell_1}\mathbf{1}^{\ell_2}\mathbf{0}^{\ell_3}\ldots$. (We can determine which case by a single additional bit.) Using run-length encoding, we cannot require less than

$$E(L) = \sum_{i=1}^{j} lb(\ell_i) \tag{3.2}$$

bits. By a similar argument to Fact 1, we can prove the following:

Fact 2. *For a conceptual bitvector B_D of known length n, such that B_D ends with a* **1***, its run-length encoding L satisfies $E(L) < E(G) + t$, where $t \leq n/2$ is the number of* **1***s in B_D.*

Proof. We first consider the case where we encode each run of **1**s in unary encoding, i.e., we encode each **1** using one bit. In total, the t **1**s require t total bits. We encode each run ℓ of **0**s in $lb(\ell)$ bits; thus, the encoding of **0**s is unchanged. (Note that this scheme is still decodeable when the γ code is used instead of lb, since there are no zero-length runs and γ codes begin with **0**.) It is plain to see that $E(L) \leq E(G) + t$. If we change our encoding of **1**s to use lb instead of unary, encoding the runs of **1**s will certainly take no more than t bits, thus proving the fact. $\square$

We do not claim that $E(G)$ or $E(L)$ is the minimal number of bits required to store D. For instance, storing the even numbers in B_D implies that $\ell_i = 1$ (for all i), and thus $E(L) \approx \lg \binom{n}{t} \approx 2t = n$. Using RLE twice to encode B_D, we obtain $O(\lg n)$ required bits, as indicated by Kolmogorov complexity. On the other hand, finding the Kolmogorov complexity of an arbitrary string is undecidable [LV97].

Despite its theoretical misgivings, we give experimental results on random data in Table 3.1 showing that $E(L) \leq \lg \binom{n}{t}$. Data generated are bitvectors B_D whose gap encoding G is produced by choosing a maximum gap length and generating uniformly random gaps in G between 0 and that maximum length (reported on a logarithmic scale in the first column). The second column, denoted RLE+γ, reports the average number of bits per gap (bpg) required to encode B_D using RLE to generate L and the γ code to encode the integers in L, as described before. The third column, denoted Gap+γ, reports the average number of bits per gap required to encode B_D using the gaps in G represented with the γ code. The fourth column reports the value of $\lg \binom{n}{t}$, where n is the length of B_D and t is the number of **1**s in it. Since t is also the number of gaps in G, the figure is still the average number of bits per gap. In the last two columns, we report similar results for the average number of bits per gap in $E(L)$ and $E(G)$.

$\lg(gap)$	RLE+γ	Gap+γ	$\lg\binom{n}{t}$	$E(L)$	$E(G)$
1	1.634	2.001	1.378	1.315	1.500
2	2.900	3.000	2.427	2.199	2.000
3	4.477	4.000	3.439	3.111	2.500
4	6.256	5.625	4.442	3.998	3.313
5	8.142	7.374	5.445	5.000	4.187
6	10.091	9.193	6.440	5.995	5.097
7	12.067	11.116	7.443	6.993	6.058
8	14.075	13.073	8.444	7.989	7.037
9	16.056	15.030	9.444	8.990	8.015
10	18.124	17.029	10.449	10.004	9.014

Table 3.1: Comparison between RLE encoding (RLE+γ), gap encoding (Gap+γ), and related measures ($\lg\binom{n}{t}$, $E(L)$, and $E(G)$). Each bitvector B_D is produced by choosing a maximum gap length and generating uniformly random gaps of **0**s between consecutive **1**s. The gap column indicates the maximum gap length on a logarithmic scale. The values in the table are the bits per gap (bpg) required by each method.

$E(L)$ outperforms $\lg\binom{n}{t}$ for real data sets, since the worst case for RLE (all equally spaced **1**s) hardly occurs. We also observe that RLE+γ outperforms Gap+γ for small gap sizes (namely 4 or less). This behavior motivates our choice for RLE to implement succinct dictionaries (in the context of compressed text indexing), since many gap sizes are small in our distributions.

3.2.2 Empirical Distribution of RLE Values and γ Codes

To validate our choice of using RLE+γ encoding, we generated real data sets for succinct dictionaries and performed experiments, comparing the space occupancy of several different encodings instead of the γ code. We took text files from the Canterbury and Calgary Corpora [Can], obtained their Burrows-Wheeler transform (`bwt`), performed the wavelet tree construction on the `bwt` according to the text indexing structure of [GGV03], and recorded the sets of integers that need to be stored succinctly. On these sets, we ran the experiments summarized in Table 3.2 and Table 3.3. We measured the total amount of bits required by every encoding for each text file and divided that amount by the length of each file; hence, the values in the tables are the bits per symbol (bps) required by each encoding method.

For Table 3.2, each encoding scheme is used in conjunction with RLE to provide the results in the table. (We also report Gap+γ for comparison purposes.) Gol refers to the Golomb code, and uses the median value as its parameter b. Manis refers to the Maniscalco code [Nel] that is tailored for use with RLE in `bwt`. Ber is the skewed Bernoulli model with the median value as its parameter b. MixBer uses just one bit to encode gaps of length 1, and for other gap lengths, it uses one bit plus the Ber code. This experiment shows that the underlying distribution of gaps in our data is Bernoulli. (When $b = 1$, the skewed Bernoulli code is equal to γ.) Notice that, except for `random.txt`, γ codes are less than 1 bps from $E(L)$. For random text, γ codes do not perform as well as expected. $E(G)$ and Gap+γ outperform their respective counterparts on `random.txt`, which represents the worst case for RLE. Finally, we do not get improved results by using RLE and δ codes as shown in Table 3.2, namely just $E(L) + \sum_{i=1}^{j} \lfloor \lg \lg (2\ell_i) \rfloor$ bits by Fact 2. Although γ coding requires $2E(L) - t$ bits, it outperforms δ in practice, since γ is more efficient for small run-lengths. Table 3.2 suggests γ as best encoding to couple with RLE.

A natural question arises as to the choice of the simplistic γ encoding, since theoretically speaking, a number of other prefix codes (δ, ζ, and skewed Golomb, for instance) outperform γ codes. However, γ encoding seems extremely robust according to the experiments above. We consider further comparisons with fractional coding and Huffman prefix codes [WMB99] in Table 3.3. In the table, the fourth column reports the bps required for the γ code in which any run-length other than 1 is encoded using γ, whereas a sequence of s **1**s is encoded with the γ code for 1 followed by the γ code for s; the fifth to Moffat's arithmetic coder in Section 3.2.3; the sixth column refers to the Huffman code in which the cost of encoding

File	$E(L)$	$E(G)$	RLE+γ	Gap+γ	RLE+δ	Gol	Manis	Ber	MixBer
`book1`	1.650	2.736	2.597	3.367	2.713	20.703	20.679	2.698	2.721
`bible.txt`	1.060	2.432	1.674	2.875	1.755	15.643	16.678	1.726	1.738
`E.coli`	1.552	1.591	2.226	2.190	2.520	2.562	2.265	2.448	2.238
`random.txt`	5.263	4.871	8.729	6.761	8.523	25.121	18.722	8.818	8.212

Table 3.2: Comparison of various coding methods when used with run-length (RLE) and gap encoding for each file listed. Unless stated otherwise, the listed coding method is used with RLE. The files indicated are from the Canterbury Corpus [Can]. The values in the table are the bits per symbol (bps) required by each method.

the (large!) prefix tree is not counted (which explains its size being smaller than that of the arithmetic code). The last two columns refer to the rangecoder mentioned in Section 3.2.3, where we employ either a fixed slack parameter $a = 0.88$ or choose the best value of a adaptively. These results reinforce the observation that γ encoding is nearly the best. In Section 3.2.3, we formalize this experimental finding more clearly by curve-fitting the distribution implied by γ onto the distribution of the run-lengths.

Improving upon γ to encode these RLE values requires a significant amount of work with more complicated methods. For the purposes of illustration, consider the comparison of γ encoding to that of an optimal Huffman encoding, given in Table 3.3. The γ code differs from Huffman encoding by at most 0.1 bps (except for `random.txt`, where the difference is 0.8 bps), and as such, this means that the majority of RLE values are encoded into codewords of roughly the same length by both Huffman and γ encoding. This news is both encouraging and discouraging. It seems that there is no real hope to improve upon γ using prefix codes, since Huffman codes are optimal prefix codes [WMB99]. Further improvement then, in some sense, necessitates more complicated techniques (such as arithmetic coding), which have their own host of difficulties, most often a greatly increased encoding/decoding time.

File	γ	δ	γ+escape	arithm.	Huffman	$a = 0.88$	adaptive a
`alice29.txt`	2.3527	2.5816	2.5934	2.4964	2.3296	2.3247	2.3272
`asyoulik.txt`	2.6304	2.9104	2.9129	2.7324	2.5946	2.5875	2.5873
`bible.txt`	1.6109	1.7677	1.7839	1.8190	1.5963	1.5901	1.5903
`cp.html`	2.6949	2.9554	2.9310	2.7170	2.6487	2.6465	2.6543
`fields.c`	2.4387	2.6145	2.5894	2.4645	2.3228	2.4186	2.4186
`grammar.lsp`	2.8121	3.0636	2.9948	2.9282	2.6694	2.7648	2.7648
`kennedy.xls`	1.4269	1.6051	1.4718	1.6834	1.3521	1.3998	1.3968
`lcet10.txt`	2.0933	2.2902	2.3047	2.1727	2.0736	2.0650	2.0684
`plrabn12.txt`	2.4686	2.7469	2.7521	2.6591	2.4354	2.4277	2.4269
`ptt5`	0.7731	0.8600	0.8617	0.9983	0.7613	0.7582	0.7580
`random.txt`	6.7949	7.9430	7.7460	6.1273	6.0004	6.5210	6.4187
`sum`	2.9500	3.2324	3.1803	2.9184	2.8765	2.8792	2.8698
`world192.txt`	1.4699	1.5890	1.6095	1.5815	1.4555	1.4540	1.4550
`xargs.1`	3.3820	3.7303	3.6564	3.3763	3.3068	3.3404	3.3404

Table 3.3: Comparison of various coding methods when used with run-length (RLE) encoding. The files indicated are from the Canterbury and Calgary Corpora [Can]. The values in the table are the bits per symbol (bps) required by each method.

3.2.3 Statistical Evidence Justifying γ Codes

We motivate our choice of γ encoding more formally, with statistical evidence suggesting that the underlying distribution of RLE values matches the distribution that the γ code (or equivalently Bernoulli, with $b = 1$) encodes optimally. For instance, consider the empirical cumulative distribution of the RLE values for `bible.txt`, shown in Figure 3.1. This distribution is fitted by the function

$$cdf(x) = e^{-a/x} \qquad x \in \mathbf{N}^+, \tag{3.3}$$

where parameter $a \in \mathbf{R}^+$ is a constant depending on the data file. For instance, in the Canterbury Corpus, we observe that $a \in [0.5, 1.8]$, depending on the file (e.g., $a = 0.9035$ for `bible.txt`). We compute the derivative of *cdf* as if it were a continuous function and

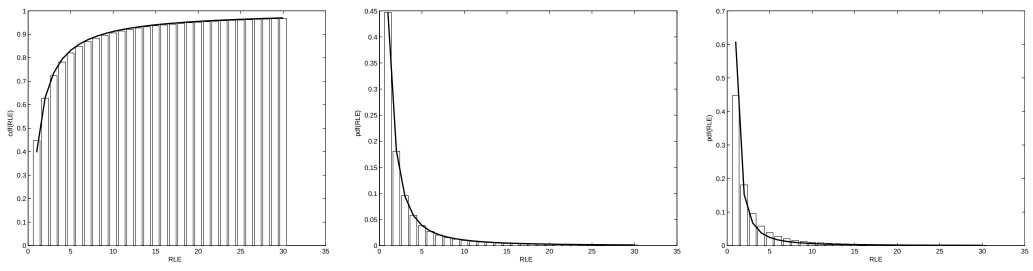


Figure 3.1: The x axis shows the distinct RLE values for `bible.txt` in increasing order. Left: The empirical cumulative distribution together with our fitting function *cdf* from (3.3). Center: The empirical probability density function together with our fitting function *pdf* from (3.4). Right: The empirical probability density function together with the fitting function $\frac{6}{\pi^2 \cdot x^2}$, where $\frac{6}{\pi^2} = \frac{1}{\sum_{i=1}^{\infty} 1/i^2}$ is the normalizing factor.

we obtain the probability density function

$$pdf(x) = \left(\frac{ae^{-a/x}}{x^2}\right) \Big/ \left(\sum_{i=1}^{\infty} \frac{ae^{-a/i}}{i^2}\right), \qquad i, x \in \mathbf{N}^+, a \in \mathbf{R}^+ \tag{3.4}$$

where the term $\sum_{i=1}^{\infty} \frac{ae^{-a/i}}{i^2}$ is the normalization factor. As one can see from Figure 3.1, function (3.4) fits the empirical probability density of the RLE values for `bible.txt` extremely well, suggesting that approximating the *cdf* by a continuous function incurs negligible error.[1]

Since $pdf(x) \sim \frac{1}{x^2}$ as x approaches infinity, we have

$$\lim_{x \to \infty} e^{-a/x} = 1 \Rightarrow \left(\frac{ae^{-a/x}}{x^2}\right) \Big/ \left(\sum_{i=1}^{\infty} \frac{ae^{-a/i}}{i^2}\right) \approx \frac{1}{x^2}.$$

Since the γ code is optimal for distributions proportional to $1/x^2$, we finally have some reasonable motivation for the success of the γ code on an RLE stream. However, these results only indicate the measure of success on prefix codes; encodings which can assign fractional bits may yet yield significant improvement.

[1]We employed the MATLAB function called `LSQCurvefit`, which finds the best fitting function in terms of the least square error between the function and the raw data to be approximated.

We performed various tests with Moffat's implementation of an arithmetic coder,[2] but the results were not satisfying when compared with the γ code. To resolve this problem, we use the statistical model of *cdf* to tailor an arithmetic coder to perform well on RLE values. Recall that both *pdf* and *cdf* depend on the knowledge of the parameter a in formula (3.3), which in turn depends on the file being encoded. (We ran experiments with a fixed $a = 0.88$, which also yielded good results on most files that we tested.) To this end, we take a fast (and free) arithmetic-style coder used in `szip` called range coder [Sch]. We encode the RLE length ℓ by assigning it an interval of length $cdf(\ell+1) - cdf(\ell) = pdf(\ell)$.[3] With this kind of compressor, we improve the compression ratio by 1–5% with respect to γ encoding. (See Table 3.3 for the comparison.) We then transform our arithmetic compressor so that the parameter a could be changed adaptively during execution, hoping for a better compression ratio. We need a cue to infer a from the values already read, so we use a maximum likelihood estimation (MLE) algorithm.

The main hurdle to simply using a maximum likelihood estimator (MLE) is its assumption of independent trials. (In our terminology, this assumption would imply that each run-length ℓ is independently drawn from its pdf.) We compute the (normalized) autocovariance of the RLE values to get an idea of "how independent" our RLE values are. This method is widely adopted in signal theory [AUT] as a good indicator of independence of a sequence of values, though it does not necessarily imply independence. In our case, the correlation between consecutive RLE values is very low for the files in Canterbury corpus, which again, though it does not imply independence in the strict sense, is a strong indication nonetheless. With this observation in mind, we assume statistical independence of the RLE values in order to define the likelihood function

[2]The code (written in Java at `<http://mg4j.dsi.unimi.it>`) is inspired by the arithmetic coder of J. Carpinelli, R. M. Neal, W. Salamonsen, and L. Stuiver, which is in turn based on [MNW98].

[3]This encoding appears to be faster than using the cumulative counts of the frequency of values already scanned, like other well-known arithmetic coders.

$$l_x(a, x_1, \ldots, x_k) = \prod_{i=1}^{k} pdf(x_i) = \left(\prod_{i=1}^{k} \frac{ae^{-a/x_i}}{x_i^2} \right) \left(\sum_{i=1}^{\infty} \frac{ae^{-a/i}}{i^2} \right)^{-k}.$$

We want to find the value of a where l_x reaches its maximum. Equivalently, we can find the maximum of $\lg l_x(a, x_1, \ldots, x_k) = L_x(a, x_1, \ldots, x_k)$. We differentiate L_x with respect to a and get

$$-\frac{\partial}{\partial a} \lg \left(\sum_{i=1}^{\infty} \frac{e^{-a/i}}{i^2} \right) = \frac{1}{k} \sum_{i=1}^{k} \frac{1}{x_i} = H(x)^{-1},$$

where $H(x)$ is the Harmonic mean of the sequence x. By denoting the left hand term by $f(a)$, we have $a = f^{-1}\big(H(x)^{-1}\big)$. Unfortunately, $f(\cdot)$ is not an analytical function and is very difficult to compute, even for fixed a. For instance, when $a = 0$, we have $f(a) = \frac{\zeta(3)}{\zeta(2)} = 0.7307629$, where $\zeta(\cdot)$ is the Riemann Z function. We apply numerical methods to approximate the function for $a \in [0.5, 1.8]$ (which is the range of interest for us). Surprisingly, all this work leads to a small improvement with respect to the non-adaptive version (where $a = 0.88$). Looking again at Table 3.3, the improvement is negligible, ranging from 1–2% at best. The best case is the file `random.txt` (in the Calgary corpus), for which the hypothesis of independence of RLE values holds with high probability by its very construction.

3.2.4 Fast Access of Experimental-Analysis-Driven Dictionaries

In this section, we focus on the practical implementation of our scheme that encodes the conceptual bitvector B_D by RLE+γ encoding and uses additional directories on this encoding to support fast access. In particular, we propose a simplified version that exploits the specific distribution of run-lengths when dictionaries are employed for text indexing purposes. Our dictionaries support *rank* and *select* primitives in $O(\lg t)$ time (with a very small constant) to obtain low space occupancy for our dictionary D seen as a bitvector B_D (with t **1**s). We represent B_D by the vector of run-length values $L = \{\ell_1, \ell_2, \ldots, \ell_j\}$ (with $j \le 2t$ and $\sum_i \ell_i = n$) , where either $B_D = \mathbf{1}^{\ell_1}\mathbf{0}^{\ell_2}\mathbf{1}^{\ell_3}\ldots$ or $B_D = \mathbf{0}^{\ell_1}\mathbf{1}^{\ell_2}\mathbf{0}^{\ell_3}\ldots$. (We use a single extra bit to denote which case occurs.)

(1) Let $\gamma(x)$ denote the γ code of the positive integer x. We store the stream $\gamma(\ell_1) \cdot \gamma(\ell_2) \cdots \gamma(\ell_j)$ of encoded run-lengths. We store the stream in double word-aligned form. Each portion of such an alignment is called a *segment*, is parametric, and contains the maximum number of consecutive encoded run-lengths that fit in it. We pad each segment with dummy **1**s, so that they all have the same length of $O(1)$ words. (This padding adds a total number of bits which is negligible.) Let $S = S_1 \cdot S_2 \cdots S_k$ be the sequence of segments thus obtained from the stream.

(2) We build a two-level (and parametric) directory on S for fast decompression.

- The bottom level stores $|S_i|^{\mathbf{0}}$ and $|S_i|^{\mathbf{1}}$ for each segment S_i, where $|S_i|^{\mathbf{0}}$ (respectively, $|S_i|^{\mathbf{1}}$) denotes the sum of run-lengths of **0**s (respectively, **1**s) relative to S_i. We store each value of the sequence $|S_1|^{\mathbf{0}}, |S_1|^{\mathbf{1}}, |S_2|^{\mathbf{0}}, |S_2|^{\mathbf{1}}, \ldots, |S_k|^{\mathbf{0}}, |S_k|^{\mathbf{1}}$ using byte-aligned codes with a continuation bit. We then divide the resulting encoded sequence into groups $G_1, G_2, \ldots, G_m$, each group containing several values of $|S_i|^{\mathbf{0}}$ and $|S_i|^{\mathbf{1}}$ for consecutive values of i. The size of each group is $O(1)$ words.
- The top level is composed of two arrays ($A_{\mathbf{0}}$ for **0**s, and $A_{\mathbf{1}}$ for **1**s) of word-aligned integers. Let $|G_j|^{\mathbf{0}}$ (respectively, $|G_j|^{\mathbf{1}}$) denote the sum of run-lengths of **0**s (respectively, **1**s) relative to G_j. The ith entry of $A_{\mathbf{0}}$ stores the prefix sum $\sum_{j=1}^{i} |G_j|^{\mathbf{0}}$. The entries of $A_{\mathbf{1}}$ are similarly defined. We also keep an array of pointers, where the ith pointer refers to the starting position of G_i in the byte-aligned encoding at the bottom level (since the first two arrays can share the same pointer). To perform the binary search in $A_{\mathbf{0}}$ or $A_{\mathbf{1}}$, we require $O(\lg t)$ time. All other work (accessing the array of pointers and traversing the bottom level) is $O(1)$ time.

The implementation of *rank* and *select* follows the same algorithmic structure. For example, to compute $select_{\mathbf{1}}(x)$ we perform a binary search in $A_{\mathbf{1}}$ to find the position j of the predecessor $x' = A_{\mathbf{1}}[j]$ of x. (Interpolation search does not help in practice to get $O(\lg \lg t)$ expected time in this case.) Then, using the jth pointer, we access the byte-aligned codes for group G_j and scan G_j sequentially with partial sums looking at $O(1)$ $|S_i|^{\mathbf{0}}$

and $|S_i|^{\mathbf{1}}$ values until we find the position of the predecessor x'' for $x - x'$ inside G_j. At that point, a simple offset computation leads to the correct segment S_i (due to our padding with dummy bits). We scan the $O(1)$ words of S_i to find the predecessor of $x - x' - x''$ in S_i. We accumulate the partial sum of bits that are to the left of this predecessor. This sum is the value to be returned as $\mathit{select}_{\mathbf{1}}(x)$. In *rank*, we reverse the role of the partial sums in how they guide the search, but the search is largely the same.

As should be clear, the access is constant-time except for the binary search in $A_{\mathbf{0}}$ or $A_{\mathbf{1}}$. In Section 3.3, we will organize many of these dictionaries into a tree of dictionaries, performing a series of *select* operations along an upward traversal of p nodes/dictionaries in the tree. Since we need to perform a binary search in each of these p dictionaries, we obtain a cost of $O(p \lg t)$ time. This cost is prohibitive: we now describe a method to reduce the time to $O(p + \lg t)$ using an idea similar to fractional cascading [CG86].

Suppose dictionary D is the child of dictionary D' in the tree. Suppose also that we have just performed a binary search in $A_{\mathbf{0}}$ of D. We can predict the position in $A_{\mathbf{0}}$ of D' to continue searching. So instead of searching from scratch in $A_{\mathbf{0}}$ of D', we retain a shortcut link from D to indicate the next place to search in $A_{\mathbf{0}}$ of D', with a constant number of additional search steps. Thus, the binary search in p dictionaries along a path in the tree will be costly only for the first node in the path (the root). This approach requires an additional array of pointers for the shortcut links, though as we will show in Section 3.4.4, the additional space required can be made negligible in practice.

3.3 Review of Wavelet Trees

In this section, we review the wavelet tree from Section 2, which forms the basis for both our indexing and compression methods. The *wavelet tree* reduces the redundancy inherent in maintaining separate dictionaries for each symbol appearing in the text; each successive dictionary only encodes those positions not already accounted for previously. Encoding the dictionaries this way achieves the high-order entropy of the text. However, the lookup

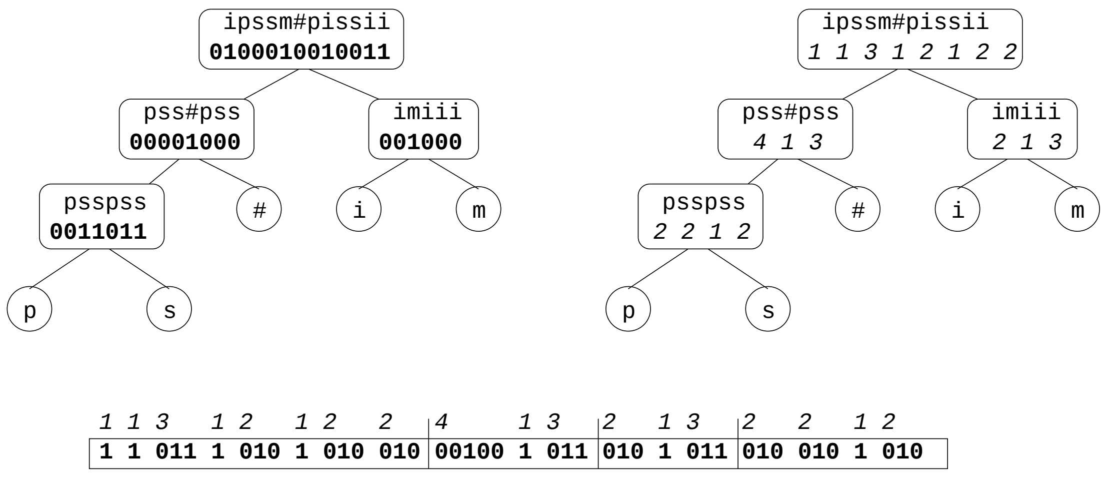


Figure 3.2: Left: an example wavelet tree. Right: an RLE encoding of the wavelet tree. Bottom: actual encoding in memory of the right tree in heap layout with γ encoding.

time for a particular item could be linear in the number of dictionaries, as a query must backtrack through all the previous dictionaries to reconstruct the answer. The *wavelet tree* relates a dictionary to an exponentially growing number of dictionaries, rather than simply all prior encoded dictionaries. Consider the example wavelet tree in Figure 3.2 (which we have augmented to explain some practical considerations as well), built on the `bwt` of the text `mississippi#`, where `#` is an end-of-text symbol.

We implicitly associate each left branch with a **0** and each right branch with a **1**. Each internal node u is a dictionary with the elements in its left subtree stored as **0**, and the elements in its right subtree stored as **1**. For instance, consider the leftmost internal node in the left tree of Figure 3.2, whose leaves are `p` and `s`. The dictionary (aside from the leading **0**) indicates that a single `p` appears in the `bwt` string, followed by two `s`'s, and so on. We don't actually store the leaves of the wavelet tree; we have included them here for clarity. The second tree indicates an RLE encoding of the dictionaries, and the bottom bitvector indicates its actual storage on disk in heap layout with a γ encoding of the run-lengths described previously. The leading **0** in each node of the wavelet tree creates a unique association between the sequence of RLE values and the bitvector.

Since there are at most $|\Sigma|$ dictionaries (one per symbol), any symbol from the text can be decoded in just $O(\lg |\Sigma|)$ time by using a balanced wavelet tree. This functionality is also sufficient to support multikey *rank* and *select*, which we support for any symbol $c \in \Sigma$. See [GGV03] for further discussion of the wavelet tree.

We introduce two improvements for further speeding up the wavelet tree—use of fractional cascading and adoption of a Huffman prefix tree shape. First, we implement shortcut links for fractional cascading as described at the end of Section 3.2.4. Second, we minimize access cost to the leaves by rearranging the wavelet tree. One can prove that theoretically, the space occupancy of the wavelet tree is oblivious to its shape [GGV03]. (We defer the details of the proof in the interest of brevity, though the reader may be satisfied with the observation that the linear method of evaluating dictionaries is nothing more than a completely skewed wavelet tree.)

We performed experiments to verify the truth of this theoretical observation in practice. Briefly, we generated $10,000$ random wavelet trees and computed the space required for various data. Our experiments indicated that a Huffman tree shape was never more than 0.006 bps more than any of our random wavelet trees. Those savings were less than a 0.1% improvement in the compression ratio with respect to the original data. Most generated trees (over 90%) were actually worse than our baseline Huffman arrangement, and did not justify the additional computation time.

Since the shape does not seem to affect the space required, we can organize the wavelet tree to minimize the access cost (for instance), under the assumption that the distribution of calls to the wavelet tree is known a priori. To describe the above more formally, let $f(c)$ be the estimated number of accesses to leaf $c \in \Sigma$ in the wavelet tree (which again is not stored explicitly). We build an optimal Huffman prefix tree by using $f(c)$ as the probability of occurrence for each c. It is well-known that the depth of each leaf is at most $1+\lg \sum_x f(x)/f(c)$, which is nearly the optimal average access cost to c. Thus, on average, we require $1 + \lg \sum_x f(x)/f(c)$ calls to *rank* or *select* involving leaf c.

Lemma 27. *Given a distribution of accesses to the wavelet tree in terms of the estimated*

Huffman	Cascading	`bible.txt`	`book1`
No	No	1.344	1.249
No	Yes	1.269	1.296
Yes	No	1.071	0.972
Yes	Yes	1.000	1.000

Table 3.4: Effect on performance of wavelet tree using fractional cascading and/or a Huffman prefix tree shape. The columns for Huffman and Cascading indicate whether that technique was used in that row. The values in the table represent a ratio of performance normalized with the case in the last row. (Lower numbers are better.)

number $f(c)$ of accesses to each leaf c, we can shape it so that the average access cost to leaf c is at most $1 + \lg \sum_x f(x)/f(c)$. The worst-case space occupancy of the wavelet tree does not change as a result of this change of shape.

In the experiments below, we make the empirical assumption that $f(c)$ is the frequency of c in the text (other metrics are equally suitable as seen in Lemma 27), reducing the weighted average depth of the wavelet tree to $H_0 \leq \lg |\Sigma|$. We performed experiments to demonstrate the effectiveness of fractional cascading and the Huffman-style tree shaping. Some results are summarized in Table 3.4. Each row contains one of the four possible cases indicating whether Huffman (first column) and fractional cascading (second column) were used. The last two columns report the corresponding timings for two text files, obtained by decompressing the entire file using repeated calls to the wavelet tree. This method is not the most efficient way to decompress a file, but it does give a good measure of the average cost of a call to the wavelet tree. Timings are normalized with the case in the last row. As can be seen from the data, fractional cascading does not always improve the performance, while Huffman shaping gives a respectable improvement.

The resulting wavelet tree is itself an index that achieves 0-order compression and allows decoding of any symbol in $O(H_0)$ expected time. In particular, it's possible to decompress any substring of the compressed text using just the wavelet tree. This structure is a perfect example where indexing *is* compression. We performed some experiments to evaluate the 0-order compression of `wave`, obtained by using the RLE+γ encoding with the wavelet tree. We do not add additional structures supporting fast access in `wave`.

We obtained the figures reported in Table 3.5 for some text files from the Canterbury and Calgary Corpora [Can], and some new files available on TREC Tipster 3 [Tip]. Our results for `wave` are in the second column. The arithmetic code [RL79] gives better results than `wave` when run on the same files, as reported in the third column `arit`. The next five columns report the figures for other compressors on the same files. In these columns, `bzip2` version 1.0.2 is the Unix implementation of block sorting based on the Burrows-Wheeler transform; `gzip` is version 1.3.5; `lha` is version 1.14i [lha]; and `vh1` is Karl Malbrain and David Scott's implementation of Jeffrey Scott Vitter's dynamic Huffman codes; `zip` is version 2.3. Note that a direct comparison of the methods may not be meaningful in some cases because of different parameters; for example, `bzip2` works on blocks of 900Kb and `book1` is the only file within this size (768771 bytes). The purpose of Table 3.5 is to show that `wave` is not particular efficient as a 0-order compressor when applied directly to a text file. Surprisingly, when applied to the `bwt` stream obtained from that file (denoted `wzip`), its performance improves a lot with respect to `wave`, as shown in the last column of Table 3.5.

The lesson learned so far suggests that the wavelet tree, coupled with RLE and γ encoding, is a simple but effective means for compressing the output of block-sorting transforms such as `bwt`.

3.3.1 Efficient Construction of the Wavelet Tree

In this section, we discuss efficient methods of constructing our wavelet tree. In particular, we detail an algorithm to create the wavelet tree in just $O(n + \min(n, nH_h) \times \lg |\Sigma|)$ time. Directories that enable fast access to our wavelet tree can be created in the same time.

File	`wave`	`arit`	`bzip2`	`gzip`	`lha`	`vh1`	`zip`	`wzip`
`book1`	**5.335**	4.530	2.992	2.953	2.967	4.563	2.954	**2.619**
`bible.txt`	**5.004**	4.309	1.931	1.941	1.939	4.353	1.941	**1.631**
`E.coli`	**2.248**	2.008	2.189	2.337	2.240	2.246	2.337	**2.181**
`world192.txt`	**5.572**	3.043	1.736	1.748	1.743	5.031	1.749	**1.519**
`ap90-64.txt`	**5.392**	4.913	2.189	2.995	2.862	4.938	2.995	**1.668**

Table 3.5: Wavelet tree with RLE+γ encoding as a plain 0-order compressor (column `wave`) and applied to the `bwt` stream (column `wzip`). Remaining columns are for other compressors. The values in the table are in bits per symbol (bps).

We can add these directories to our `wzip` format for fast access. We now describe `wzip` in detail. The header for `wzip` contains three basic pieces of information: the text length n, the block size b, and the alphabet size Σ. The body of the encoding is then $\lceil n/b \rceil$ blocks, each block encoding b contiguous text symbols (except possibly the last block). Recall that the nodes of the wavelet tree are stored in heap ordering (example in Figure 3.2). We break this stream into blocks and encode it. The format for a block is given below:

- A (possibly compressed) bitvector of $|\Sigma|$ bits that stores the symbols actually occurring in the block. Let $\sigma \leq |\Sigma|$ be the number of symbols present. (For large Σ, we may store the bitvector in the header, with smaller bitvectors in the blocks that refer only to the symbols stored in the bitvector in the header).
- The dictionaries encoded with RLE+γ, concatenated together according to heap order. The wavelet tree has σ implicit leaves and $\sigma-1$ internal nodes with dictionaries. (See Figure 3.2 for an example.)

We do not need to store the length of each encoding, as it is already implicitly encoded. When processing, the encoding for the root node of the wavelet tree ends when the sum of the encoded RLEs equals n. (These run-lengths may be spread over several blocks.) At this point, we know the total number of **0**s and **1**s, plus the (dummy) leading **0**. The number of **0**s is the sum of the RLE values in the left child of the root, and the number of **1**s is the sum of the RLE values in the right child of the root. We can go on recursively this way, down to the implicit leaves, from which we can infer the frequency of the occurrences of each symbol in the block.

3.3.2 Compression with `bwt2wzip`

In this section, we describe our compression method `bwt2wzip`, which takes as input the `bwt` stream (the Φ function in [GGV03]) of the file and compresses it efficiently using our wavelet tree techniques. Our approach introduces a novel method of creating the wavelet tree in just $O(n + \min(n, nH_h) \times \lg |\Sigma|)$ time, which is also faster in practice, as the entropy factor can significantly lower the time required. This behavior relates the speed of compression to the compressibility of the input. Thus, we introduce a new consideration into the notion of compressibility—highly compressible data should be easier to handle, both in terms of space and time.

If we were to build the wavelet tree naively from the `bwt` stream, we would run multiple scans on the `bwt` to set up the bitvector in each individual node of the wavelet tree. Then, we would compress the resulting dictionaries with RLE+γ encoding. A single-scan method is made possible by placing one item at a time in each of the internal nodes from its root-to-leaf path via an upward walk. Given any internal node in the tree, the set of values stored there are produced in increasing order, without explicitly creating the corresponding bitvector. Since processing each symbol in the `bwt` could take up to $O(\lg |\Sigma|)$ time, it requires $O(n \lg |\Sigma|)$ time in total. We describe a refinement of this construction method requiring $O(n + \min(n, nH_h) \times \lg |\Sigma|)$ time. This method is faster in practice, since the entropy factor can significantly lower the time required for compressible text.

Let c be the current symbol in the `bwt` stream, and let u be its corresponding leaf in the wavelet tree. (Recall that the numbering of internal nodes follows the heap layout.) While traversing the upward path in the wavelet tree to the root, we decide whether the run of bits in the current node should be extended or switched (from **0** to **1** or vice versa). However, we do not perform this task individually for each symbol. Instead, we process consecutive runs of equal symbols c, say r_c in number, in the input simultaneously. We then extend the runs in each internal node of the wavelet tree r_c units at a time. Let n_r be the number of such runs that we process for the entire `bwt` stream.

To make things more concrete, we use the following auxiliary information to compress the input string `bwt`. Notice that the leaves of the wavelet tree are not explicitly represented; given a symbol $c \in \Sigma$, it suffices to know its leaf number `leaf`$[c]$. We also allocate enough space for the dictionaries `dict`$[u]$ of the internal nodes u. We keep a flag `bit`$[u]$ for each internal node u, which is **1** if and only if we are currently encoding a run of **1**s in u. Below, we describe and comment the main loop of the compression. We do not specify the task of encoding the RLE values with γ codes, as it is a standard computation performed on the dictionaries `dict`$[u]$ of the internal nodes u.

```
1 while ( bwt != end ) {
2   for ( c = *bwt, r_c = 1; bwt != end && c == *(++bwt); r_c++ ) ;
3   u = leaf[c];
4   while ( u > 1 ) {
5     if ( (u & 0x1) != bit[u >>= 1] ) {
6        bit[u] = 1 - bit[u]; *(++dict[u]) = 0; }
7     *(dict[u]) += r_c;
8   }
9 }
```

We scan the input symbol c from the current position in the `bwt` to determine r_c, the length of the run of c (line 2). We determine the heap number of the (virtual) leaf u associated with c (line 3) and start an upward traversal (lines 4–7). We close the run in the current node u and start a new run in the following two cases:

1. We arrive from the left child of u and the current run in u is made up of **1**s; or
2. We arrive from the right child of u and the current run in u is made up of **0**s.

We express this condition succinctly in line 5, where (`u & 0x1`) is **1** when u is a right child, and `u >>= 1` denotes u's parent whose flag `bit` indicates if the current run is of **1**s. We complement its value and prepare for the next entry in the current dictionary (line 6). We then extend the current run-length by r_c (line 7). We exit the loop at the root (when $u = 1$ in line 4).

The time required to perform these actions over the whole `bwt` input stream is $O(n)$ to scan the `bwt` stream, and $O(n_r \times \lg|\Sigma|)$, to perform the n_r traversals of the wavelet tree, taking $O(\lg|\Sigma|)$ time. It turns out that the number of runs n_r processed by our algorithm is $n_r = O(\min(n, nH_h))$, proving our bound. Since $n_r \leq n$ trivially, we show that $n_r = O(nH_h)$, thus capturing precisely the high-order entropy of the text. Note that n_r is asymptotically upper-bounded by the number of runs n_d in all of the dictionaries of the internal nodes in the wavelet tree. This bound holds, since either the beginning or the end of a run in the `bwt` stream must correspond to the beginning or the end (or vice versa) of at least one distinct run in a dictionary. (Otherwise, we could extend the run in the `bwt` stream, except possibly for the first or the last run). Thus, $n_r = O(n_d)$. Since each run length will require at least one bit to encode (i.e., $lb(\ell) \geq 1$ for any $\ell \geq 1$), we can simply bound the sum of the logarithm of their run-lengths. Theorem 16 proves that a single wavelet tree encoded with RLE+γ achieves $O(nH_h)$ bits of space, thus proving that $n_r = O(nH_h)$. The proof technique makes use of the framework in [GGV03], and is proved in Section 3.4.2.

3.3.3 Decompression with `wzip2bwt`

Decompression is a fairly straightforward task once the encoding has been done, though some care must be taken when decomposing sets of runs. The decompression algorithm first performs a downward traversal to identify the symbol c to decompress. It then performs an upward traversal, analogous to that in `bwt2wzip`, except that it decrements the RLE values by r_c, producing in output r_c instances of c. However, the value of r_c is not necessarily the last RLE value examined along this path; rather it is the minimum among them. The reason stems from the fact that the runs in the dictionaries in the internal nodes (except for the root) may correspond to a union of runs that were disjoint in the input string `bwt`.

Fortunately, the minimum value among those in an upward traversal from a leaf refers to an individual run in the `bwt` stream, and it is the value r_c.

To decompress, we use auxiliary information in `bwt2wzip`, a variable `alphabetsize` and an array `symbol`. The former denotes the actual number of symbols in the `bwt` stream; the symbols are numbered from 0 to `alphabetsize` - 1. To recover the original value, we remap them using array `symbol`. We now comment on our main loop for decoding. (Again, we do not describe how to decode the RLE values with the γ code, as it is a standard task.)

```
1 while( r_c = *(dict[u=1]) ) {
2   while ( (u = (u << 1) | bit[u]) < alphabetsize )
3     if ( *(dict[u]) < r_c ) r_c = *(dict[u]);
4   c = u - alphabetsize;
5   while ( u > 1 )
6     if ( !(*(dict[u >>= 1]) -= r_c) ) {
7         bit[u] = 1 - bit[u]; ++dict[u]; }
8   for( c = symbol[c]; r_c--; *(bwt++) = c ) ;
9 }
```

We start with the RLE value in the dictionary of the root ($u = 1$ in line 1). We perform the downward traversal (line 2), guided by the current run of **1**s or **0**s, looking at the flag `bit`$[u]$ to branch either to the left (`bit`$[u] = \mathbf{0}$) or the right (`bit`$[u] = \mathbf{1}$) in the heap layout. We also keep the minimum RLE value in r_c (line 3), as previously mentioned. When we reach a leaf, we find the rank of the symbol to decode (line 4). Note that lines 4 and 8 are the analogue of line 2 in `bwt2wzip`, except that we output symbol c after remapping it, with `symbol` in the current position indicated by the `bwt` stream. The upward traversal in lines 5–7 is similar to the downward traversal in lines 4–7 of `bwt2wzip`, except that we decrease the RLE values in the dictionaries. The time required for decompression follows the same argument as for compression.

3.3.4 Performance and Experiments for `wzip`

In this section, we discuss our experimental setup and detail our results for the speed of access of our compression algorithm. We used several platforms to test our algorithms: ATH = Athlon AMD 1GHz 512MB Linux, gcc version 3.3.2 (Debian); AXP = AMD Athlon XP 1.8GHz 512MB Linux, gcc version 3.2.2 20030222 (Red Hat Linux 3.2.2-5); PIII = Intel Pentium III 1GHz 512MB Windows XP, gcc version 3.2 (mingw special 20020817-1); PIV = Pentium IV 2GHz 1GB Windows XP, gcc version 3.2 (mingw special 20020817-1); and XEO = Intel Xeon 2GHz 2GB Linux, gcc version 3.3.1 20030626 (Debian prerelease). We drew our data from the Canterbury and Calgary corpora. The first three rows of Table 3.6 are files from those corpora; the last two rows are the concatenation of all the files in the same.

We compare our performance with a simple routine that copies the input `bwt` stream into another array. We normalize the timings of our routines with respect to this simple copy operation. We don't compare with the scan operation, as the compiler often cheats and doesn't generate code to scan for an empty loop. In our experiments, `bwt2wzip` (compression) is 2—6 times slower than a simple copy operation, and `wzip2bwt` (decompression) is 3—7 times slower. The difference in performance depends mainly on the architecture of the processor rather than the input file. (Consult Table 3.6 for proof of this fact, with bold figures for the minimum and the maximum.) The computation of RLE takes roughly 30% of the total time in `bwt2wzip` and 40% in `wzip2bwt`.

With regard to fine tuning performance in the code for `bwt2wzip` and `wzip2bwt`, each time we access an entry pointed to by `dict`$[u]$, we may initiate a cache miss. Also, we need to pre-allocate more space to accommodate all the dictionaries (whose final size is known only at the end of the compression, which is too late). We alleviate this problem by synchronizing the access to the decoded RLE values. In particular, we can provide the same access pattern during the execution of `bwt2wzip` and `wzip2bwt`. Some care must be taken at initialization to maintain this information.

Consequently, the RLE values are scrambled among the dictionaries and follow the

	`bwt2wzip`					`wzip2bwt`				
File	ATH	AXP	PIII	PIV	XEO	ATH	AXP	PIII	PIV	XEO
`ap5.txt`	4.811	2.822	2.244	4.878	5.250	6.736	4.200	3.438	6.232	6.500
`bible.txt`	4.093	2.688	2.162	3.473	4.370	5.302	3.656	2.910	4.746	5.037
`world95.txt`	3.077	2.375	**1.946**	2.705	3.800	3.744	3.167	**2.698**	3.750	4.450
`calgary`	4.465	3.481	2.566	4.162	5.565	6.256	5.148	3.939	5.643	**6.826**
`canterbury`	4.419	3.091	2.324	3.255	**5.625**	5.839	4.318	3.522	4.614	6.625

Table 3.6: Running times for `bwt2wzip` and `wzip2bwt` normalized with that of a simple copy routine. File sizes in bytes are 5,000,000 for `ap5.txt`, 4,047,392 for `bible.txt`, 2,899,483 for `world95.txt`, 3,215,493 for `calgary`, and 2,810,784 for `canterbury`.

access pattern of `wzip2bwt`. To solve this problem, we no longer keep a pointer in `dict`$[u]$; instead, we temporarily store the current RLE value for u. As a result, except for `dict`$[u]$, `bit`$[u]$, and `symbol`, access to the other structures is sequential, which enables us to exploit the many levels of cache. Moreover, we do not need to allocate temporary storage to keep the RLE values that we will encode. Rather, we can produce each RLE value and encode it on the fly. A drawback of this approach is that we lose compatibility with the text indexing functionalities in Section 3.4.

It is worth noting that the total cost of compression and decompression is much larger than what we discussed so far. We must also account for the cost of suffix sorting to obtain the `bwt` stream from the input text file (in addition to that of `bwt2wzip`) and the cost of obtaining the text file from the `bwt` stream (in addition to that of `wzip2bwt`).

3.4 Practical Suffix Arrays: Indexing Equals Compression

We explored dictionary methods which perform well in practice. Now, we apply these dictionary methods to compressed suffix arrays [GGV03, GV05, Sad03, Sad02b] and show both experimental success as well as a theoretical analysis of these practical methods. First, we provide some background notions from [GV05, GGV03].

3.4.1 Compressed Suffix Arrays (CSA)

To recap, a standard suffix array [GBS92, MM93] 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, $T\big[SA[i], n\big]$. 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 [GV05] contains the same information as a standard suffix array.

Definition 3. Given a text T of length n, a *compressed suffix array* [GV05, Sad03, Sad02b] 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;
- *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 $1 \le c \le n - SA[i] + 1$.

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. The recursive decomposition is given below:

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} (representing the recursion).

3. The final level, $\ell = \lg\lg n$ is written explicitly, using n bits.

SA_k is not explicitly stored (except at the last level ℓ), but we refer to it for the sake of explanation. B_k is a bitvector such that $B_k[i] = \mathbf{1}$ if and only if $SA_k[i]$ is even. Even-positioned suffixes are divided by 2 and represented in SA_{k+1}. 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 $p+1$. We describe it by the following formula (also handling the case when $SA_k[i] = n$):

$$\Phi_k(i) = \Big\{ \; j \quad \text{such that} \quad SA_k[j] = (SA_k[i] \bmod n) + 1 \; \Big\}. \tag{3.5}$$

A *lookup* for $SA_k[i]$ can be answered in the following way:

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

The representation of B_k and $\mathit{rank}(B_k, i)$ uses standard techniques and is easy to compress. The major hurdle for compression remains in the representation of Φ_k, which is at the heart of compressed suffix arrays and indexing in general. The key to the compression of Φ_k (which leads to a bound in terms of nH_h) is that we can partition the function Φ_k into a series of increasing subsequences (or sublists) that refer to positions in the text storing the concatenated string yx, for each symbol $y \in \Sigma$ and context $x \in P_h^*$, the optimal prefix cover [FGMS05] for contexts of length at most h. These sublists $\langle x, y \rangle$ can be stored by succinct dictionaries using $\lg \binom{n_k^x}{n_k^{x,y}}$ bits, where n_k^x is the number of suffixes of T prefixed by context x at level k and $n_k^{x,y}$ is the number of suffixes in T prefixed by the concatenated string yx at level k. Additionally, each sequence of sublists related to $yx_1, yx_2, \ldots, yx_c$, where $c = |P_h^*|$ and $x_i \in P_h^*$ is lexicographically before x_{i+1}, also forms an increasing subsequence. We call these lists Σ-lists, one for each symbol y in the text. Each dictionary is stored according to a much-reduced universe size using the wavelet tree; we refer the reader to [GGV03] for further details on the consequences of this observation with regard to compression.

3.4.2 Practical Considerations for Compressed Suffix Arrays

In this section, we apply our practical dictionaries to the CSA framework we described in Section 3.4.1, achieving practical data structures that implicitly achieve at most twice the high-order entropy of the text.

Theorem 15. *We can encode the n_k entries in all sublists at level k of the compressed suffix array using at most $2nH_h + o(n)$ bits, if we store each sublist as a succinct dictionary D using RLE+γ encoding.*

Proof. Each of our dictionaries D takes at most $E(L) + \sum \lg(g_i + 1)$ bits of space (since they are RLE+gamma dictionaries). Since $E(L) \leq E(G) + t$ by Fact 1 and $E(G) = \sum \lg(g_i + 1) + t$ by Fact 2, we can bound the size of each dictionary by $2E(G)$. Thus, we can replace our dictionaries with the ones in the analysis in [GGV03], at most doubling the theoretical worst-case bounds. The result follows automatically from the analysis in [GGV03]. $\square$

This discovery brings up a remarkable point—our practical dictionary is blind to the universe size that was so carefully constructed in [GGV03] to allow the use of the fully indexable dictionaries from [RRR02] (whose space occupancy is almost linearly dependent on the universe size).

We propose operating implicitly on any partition $P_h \subseteq \Sigma^h$ (including a partition based on the optimal prefix cover P_h^* [FGMS05]) for $h \geq 0$, where $|P_h| \leq n^\alpha$, for some $0 < \alpha < 1$. (This reasonable assumption is also used in [GGV03].) We argue that due to the nature of our directory, we are still able to achieve the higher-order entropy given in [GGV03]. Said more mathematically, we can split the cost in [GGV03] as $nH_h + M(h)$, where $M(h)$ refers to the overhead necessary to encode a statistical model for contexts of length up to h. However, the term $M(h)$ may become large for sufficiently large values of h, since we may have $nH_h = 0$ in this case.

Fact 3. *There exists an $h' < n$, such that for each $h > h'$, we have $nH_h = 0$.*

Proof. Build a suffix tree on the text terminated with n endmarkers that do not appear elsewhere. Consider one of the internal nodes storing the longest string, say of length h'. Then, for any context $h > h'$, prune the suffix tree, leaving only strings of length $h+1$. We can predict the $(h+1)$st symbol with conditional probability $p = 1$, since we are on an arc leading to a terminal node. (There are no more branches.) At this depth, every symbol can be predicted with perfect accuracy. The information content of such a distribution is 0, requiring no bits (i.e., everything is encoded in $M(h)$ bits in the model, which relates to the pruned suffix tree). Hence, $nH_h = 0$ for $h > h'$. □

In similar cases (in our experiments when $h > 4$ and for more moderate cases than Fact 3), the contribution of $M(h)$ may dominate the expression. This observation motivates the need to acknowledge the model cost as a significant factor in compression. Now we prove our main theorem in this section, which describes how to encode the Φ function in equation (3.5).

Theorem 16. *We can encode the neighbor function Φ using $2nH_h + o(n)$ bits with γ encoding, thus implicitly achieving high-order entropy.*

Proof. For ease of exposition, we "number" the lexicographically ordered symbols y as $1 \leq y \leq |\Sigma|$ and similarly number the lexicographically ordered contexts x as $1 \leq x \leq |P_h|$. Recall that each Σ list is an increasing subsequence of positions. In [GGV03], we conceptually break down the Σ lists that constitute the neighbor function Φ of compressed suffix arrays into sublists for each context of order up to h (to scale the universe size in the dictionaries). We now encode all the sublists for the same symbol in one shot using our succinct dictionaries and the wavelet tree. The difference in encoding is that we save space by not storing pointers to the beginning of each sublist (which can contribute significantly to the space $M(h)$ for the statistical model). On the other hand, our gaps can be longer when the gap we encode traverses a sublist. The idea of the proof is to show that the savings more than make up for the loss. We define the problem below formally.

Let g_j be the jth gap in list y (composed of n^y items) such that the jth item s_j in list y is in context $x_j \in P_h$ and the $(j+1)$st item s_{j+1} in list y is in context x_{j+1}, where $x_j \leq x_{j+1}$. Thus, s_j is in sublist $\langle x_j, y\rangle$ and s_{j+1} is in sublist $\langle x_{j+1}, y\rangle$. We decompose the gap g_j into three parts:

- g'_j, the length of the jump out of sublist $\langle x_j, y\rangle$;
- g''_j, the length of the jump over empty sublists inside of list y, namely a subset of the sublists $\langle x_j+1, y\rangle, \langle x_j+2, y\rangle, \ldots, \langle x_j+k, y\rangle$ where $x_j+k+1 = x_{j+1}$; and
- g''', the length of the jump within sublist $\langle x_{j+1}, y\rangle$.

By definition, $g_j = g'_j + g''_j + g'''_j$. The value g'''_j is the only non-zero quantity when s_j and s_{j+1} are in the same context x i.e., $x_j = x = x_{j+1}$. Said differently, $g_j = g'''_j$ in this case, since we are not encoding a gap that jumps over other sublists. This is the same cost incurred in [GGV03] when the sublists are treated separately (since they never encode a gap that traverses a sublist). Since $\lg g_j \leq \lg(g'_j + g''_j) + \lg g'''_j$, we can bound our total overhead by

$$\sum_{y\in\Sigma}\sum_{i=1}^{n^y-1} \lg g_j - \lg g'''_j \leq \sum_{y\in\Sigma}\sum_{i=1}^{n^y-1} \lg(g'_j + g''_j) = o(n);$$

this is exactly the additional cost we incur by treating all of our sublists together. Since we incur overhead for each sublist exactly once, taking $\lg(g'_j + g''_j) = O(\lg n)$ bits, we can bound this cost by the number of sublists among the entire structure of [GGV03]. We now give more details on bounding the above quantity. Let the number of contexts $c = |P_h| = n^\alpha$, where $0 < \alpha < 1$, the same restriction as [GGV03]. For list y, we can have at most $\min\{c, n^y\}$ items with non-zero values for g'_j and g''_j. Since $\sum_j(g'_j + g'_j) \leq n$, we can encode these gaps using a dictionary, taking $\lg\binom{n}{c} = o(n)$ bits per list. We can similarly apply the bound for each Σ list, taking at most $|\Sigma|$ times as much space, which is again $o(n)$ bits. Finally, since we are using γ encoding instead of a more efficient code, we at most double the encoding cost of each dictionary as in Theorem 15, thus doubling the entropy term and proving the claimed bound. $\square$

3.4.3 Suffix Array Compression

One major advantage of suffix sorting (block sorting) is that not only does it compress according to high-order entropy, it also concisely represents the underlying statistical model, typically exploited using a Move-to-Front (MTF) encoder [BSTW86] (as it happens in `bzip2`). We now describe how to use our succinct dictionaries (RLE+γ), the suffix array (block sorting), and the wavelet tree (incremental representation of dictionaries) to achieve a compression ratio comparable to that of methods such as `bzip2`, without using MTF, arithmetic, or multi-table Huffman encoding. (See also [WM01].) Based on our analysis, we conclude that our approach avoids explicit treatment of the order of context, but allows for indirect context merging through the run-length encoding.

The outcome of our experiments is summarized in Table 3.7, where the rows represents some text files from the Canterbury and Calgary corpora except the last ones (`ap90-64.txt`, `ap90-100.txt`), which are some news files available on TREC Tipster 3 [Tip]. Each row represents duplicated experiments performed as follows. (Figure 3.2 may help the reader.)

1. We obtain the `bwt` stream from the input text file.
2. If (MTF = Yes), we transform the `bwt` stream using MTF.
3. We build the wavelet tree on the stream resulting from the previous two steps.
4. For each bitvector B_D found in the wavelet tree, we produce the corresponding sequence L of (positive) integer run-lengths.
5. We encode the integers in the sequences L thus obtained, using one of the following encodings: γ code, δ code, Gol code, Manis code, Ber code, or MixBer code.
6. We divide the total number of bits required by the encoding in the previous step by the size of the input text file to obtain the bits per symbol (bps).

Column $E(L)$ reports the bps quantity using formula (3.2) in Section 3.2.1. We take $E(L)$ as an empirical lower bound to the figures for the other codes. (Note that the integers in L change when using MTF, as a consequence of step 2.) The last six columns of Table 3.7 report the resulting bps figures for the γ, δ, Gol, Manis, Ber, and MixBer codes. Gol refers to the Golomb code, and uses the median value as its parameter b; Manis refers to code [Nel]; Ber is the skewed Bernoulli model with the median value as its parameter b; MixBer uses just one bit to encode gaps of length 1, and for other gap lengths, it uses one bit plus the Bernoulli code.

Table 3.7 shows that that Move-To-Front (MTF) and Huffman/arithmetic coding are not strictly necessary to achieve high-order compression in our case; see the column for the γ code for an example. Notice that Maniscalco and Golomb gain a huge savings from using MTF: We do not have an explanation for the gap between Golomb and Bernoulli without using MTF. (Golomb encodes a positive integer x using $1 + \lfloor (x-1)/b \rfloor + \lfloor \lg b \rfloor$ bits, where b is the median value in our case.) In almost all cases, the γ code performs better than any other method for each file, aside from $E(L)$.[4] In summary, we obtain high-order compression with three simple ingredients: suffix arrays, wavelet trees, and dictionaries based on RLE and γ encoding.

3.4.4 Suffix Array Functionalities

We now have all the ingredients for implementing compressed suffix arrays. We still need to store SA_ℓ and its inverse, as well as a dictionary to mark the positions in the original suffix array represented in SA_ℓ. Here we face a similar problem to that of the directories in our dictionary D where, if we follow the same techniques, we sparsify these arrays. In Table 3.8, we show the number of bits per symbol needed for compressed suffix arrays on some files from the Canterbury corpus and TREC Tipster 3 [Tip]. We incur a minimal overhead cost for adding suffix array functionality; moreover, our potentially costly fractional cascading in our wavelet tree requires almost negligible space (0.006 bps).

[4]Note that values for the γ code from Table 3.5 are larger than their corresponding (non-MTF) entries in the γ column, as the former must includes some padding bits to allow fast access.

File	MTF	$E(L)$	γ	δ	Gol	Manis	Ber	MixBer
`book1`	No	1.650	**2.585**	2.691	20.703	20.679	2.723	2.726
`book1`	Yes	1.835	**2.742**	3.022	3.070	2.874	2.840	2.921
`bible.txt`	No	1.060	**1.666**	1.740	15.643	16.678	1.742	1.744
`bible.txt`	Yes	1.181	**1.753**	1.940	2.040	1.926	1.826	1.844
`E.coli`	No	1.552	**2.226**	2.520	2.562	2.265	2.448	2.238
`E.coli`	Yes	1.584	2.251	2.566	2.445	**2.232**	2.398	2.261
`world192.txt`	No	0.950	**1.536**	1.553	19.901	21.993	1.587	1.589
`world192.txt`	Yes	1.035	**1.570**	1.707	2.001	1.899	1.630	1.643
`ap90-64.txt`	No	1.103	**1.745**	1.814	24.071	25.995	1.815	1.830
`ap90-64.txt`	Yes	1.235	**1.840**	2.031	2.148	2.023	1.915	1.935
`ap90-100.txt`	No	1.077	**1.703**	1.772	24.594	26.191	1.772	1.787
`ap90-100.txt`	Yes	1.207	**1.797**	1.985	2.104	1.982	1.870	1.890

Table 3.7: Measure of the effect of MTF on various coding methods when used with RLE. The MTF column indicates when it is used. The values in the table are in bits per symbol (bps) and the lowest per row are shown in boldface.

3.5 Space-Efficient Suffix Trees

In this section, we apply our ideas on suffix arrays and compression to the implementation of a space-efficient version of suffix trees [Kur99]. Suffix trees are at the heart of many algorithms on strings and sequences, so their full functionality is needed [Gus97b]. Thus, we support a suite of navigational, hierarchical, and search capability. From a theoretical point of view, a suffix tree can be implemented in either $O(n \lg |\Sigma|)$ bits or $|CSA| + 6n + o(n)$ bits [Sad02a], which is significantly larger than that of the compressed suffix arrays discussed before. The bottleneck comes from retaining the longest common prefix (LCP) information, which requires at least $6n$ bits [Sad02b]. As an alternative, the same information can be maintained in at least $4n$ bits to retain the tree shape of at most $2n - 1$ nodes [MRS01a],

	`book1`	`bible.txt`	`E.coli`	`world192.txt`	`ap90-64.txt`	`ap90-100.txt`
Φ overhead	0.166	0.050	0.050	0.067	0.032	0.032
Φ	2.785	1.681	2.231	1.586	1.700	1.659
CSA overhead	0.328	0.210	0.210	0.228	0.192	0.191
CSA	**2.946**	**1.841**	**2.391**	**1.747**	**1.860**	**1.818**

Table 3.8: Comparison of space required by Φ and the compressed suffix array (CSA), given in bits per symbol (bps). Overhead refers to all space other than the RLE+γ encoding for the data itself.

though there is some slowdown since LCP information is not stored explicitly.[5] In either case, a separate (compressed) suffix array is needed to encode the leaves of the suffix tree. Since LCP information encodes the internal nodes of the suffix tree, the bound reduces to less than $6n$ bits in practice. Despite our dictionaries, however, the space required for LCP information is not drastically diminished, since we are anyway encoding the internal structure of the suffix tree.

To achieve less than $6n$ bits, we employ a simple heuristic based on an arbitrarily chosen slowdown factor $S = O(\lg n)$. We implement part of the lowest common ancestor simplification introduced in [BFC04]. We use our dictionaries and sparsification of the entries, sped up with tricks to take advantage of parallelism in modern processors. Once we have this structure, we use just $O(1)$ additional words to get a representation of a suffix tree. For example, we obtain 2.98 bps (`book1`), 2.21 bps (`bible.txt`), 2.54 bps (`E.coli`), and 2.8 bps (`world192.txt`). These sizes are comparable to those obtained by `gzip`, namely, 3.26 bps (`book1`), 2.35 bps (`bible.txt`), 2.31 bps (`E.coli`), and 2.34 bps (`world192.txt`).[6] A point in favor of the compressed representation of suffix trees is that they fit in main memory for large text sizes, while regular suffix trees must resort to

[5]A recent manuscript by Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung improves over these bounds.

[6]The comparison with `gzip` is just to show that our implementation is space efficient, not a reason to replace `gzip`.

external memory techniques. A drawback is that accessing the former requires more CPU time. Nevertheless, we expect that their performance is superior when compared to regular suffix trees in external memory. Several applications have such large suffix trees, e.g., a suffix tree for the human genome.

We exploit a folklore relationship between suffix tree nodes and intervals in the suffix array, which has been used recently to devise efficient algorithms [AKO04, AASA01]. For each node u, there are two integers $1 \leq u_l \leq u_r \leq n$ such that $SA[u_l \ldots u_r]$ contains all the suffixes stored in the leaves descending from u. Thus, a node $u \equiv (u_l, u_r, \ell_u)$ is a triple of integers in our representation, where ℓ_u represents the *LCP* of the strings of the text beginning at positions $SA[u_l]$ and $SA[u_r]$. For each node u, we use this information to support the following operations:

- reaching u's parent;
- branching to u's child v by reading symbol s;
- finding the label of the edge (u, v) (with cost proportional to the length of the label);
- computing the skip value of u;
- determining the number of leaves descended from u;
- checking whether u is an ancestor of v;
- computing the lowest common ancestor of u and v;
- following the suffix link from u to v, in the style of McCreight or Weiner [Gus97b].

We use Kasai et al.'s linear-time method [KLA$^+$01] to compute *LCP* information. We modify Sadakane's method [Sad02b] to store only *LCP* values larger than $2 \lg n$; it works and compresses well. (We also explicitly store *LCP* values for a few constant-size *LCP*s to speed up searching.) We also implement the doubling technique of Farach-Colton and Bender [BFC04] to compute *LCP* information in constant time, though we can trade time to reduce the space required.

We base our algorithms on the fact that we can use *LCP* information to go from node u to node v by extending their intervals suitably and use the same information to navigate in the compressed suffix array. We defer the standard details for most operations and discuss only how to follow the suffix link from u to v.

Let $u \equiv (u_l, u_r, \ell_u)$ and $v \equiv (v_l, v_r, \ell_v)$. We use our wavelet tree to find two values u'_l, u'_r such that $v_l \leq u'_l \leq u'_r \leq v_r$. To find v_l and v_r, we observe that $lcp(SA[u'_l], SA[u'_r]) \geq \ell_v$. We perform two binary searches, one for u'_l going to the left subtree and the other for u'_r going to the right subtree. To find v_ℓ, at each step of our binary search in position i, we compute $lcp(SA[i], SA[u'_l])$ and compare it with ℓ_v. Depending on the outcome, we can decide which way to go. Since v_l is the leftmost position such that $lcp(SA[v_l], SA[u'_l]) \geq \ell_v$, we can find v_l in a logarithmic number of steps. Finding v_r is similar.

We now discuss our experimental setup for the suffix tree and suffix array applications. Many experiments were run on the machines ATH and XEO that we described in Section 3.3.4. The data sets were drawn mainly from the Canterbury corpus, TREC Tipster 3 [Tip], and electronic books from the Gutenberg project at `<http://promo.net/pg/>`.

Our source code is written in C in an object-oriented style. Our code is organized as five distinct modules, which we now describe briefly. Module `dict` implements our crucial dictionaries (Section 3.2). Module `phi` implements the wavelet tree and its use in compressed suffix arrays (Section 3.3), while module `csa` implements the compressed suffix array and related functionality (Section 3.4). Module `lcp` stores LCP information and module `st` implements suffix tree functionality, though we avoid storing any nodes explicitly (Section 3.5). The latter module requires fast decompression of symbols, access to the suffix array and its inverse, and fast computation of *LCP* information, all of which are provided in the other modules.

3.6 Conclusions

In this chapter, we develop the simple notions of run-length encoding (RLE) and γ encoding to achieve competitive compression ratios and fast compression and decompression time for both indexing and compression algorithms. (Of course, we must add the dominant cost of computing `bwt` by suffix sorting and that of inverting it.) Some independent work has also shown that compressed suffix arrays are still competing in search time [HLS+04]. The techniques we have developed are practically sound, but also grounded in solid theoretical analysis and strong notions of encoding both the data and the underlying model. Our method is tunable to the access pattern of any file, which is a property unknown in similar work on compressed indexing. While we do not claim that our software is a ready-to-use library, we intend to perform intense algorithm engineering to further tune the search time of our indexing structures, though much has already been done. We construct the index in competitive time (roughly 1-2 minutes for 64 MB of data on our test system).

Our compression algorithm `wzip` does not require any additional parameters beyond the text size, alphabet size, and block size, and is tailored to work for large alphabets, e.g., Unicode, UTF/16. Our method performs integer bit assignments and does not resort to costly computation of fractional bits, as does an arithmetic coding technique. A simple copy operation is only 2–6 times faster than our `wzip` compression, and only 3–7 times faster than our decompression. As a matter of fact, our encoding algorithm is so fast that its major bottleneck is the encoding and decoding of γ. However, the real bottleneck remains the fast computation of the `bwt`, namely by suffix sorting.

Despite these observations, data in `http://www.maximumcompression.com` shows that our method does not achieve the best compression ratio on the market. On the other hand, our ideas are easy to implement, as they use introductory material on standard compression techniques. Our wavelet encoding is in some sense related to inversion coding [Deo02], though the analysis in [GGV03] is the first to truly understand its impact. More critically, however, the wavelet tree serves as a vast improvement in access time over inversion coding ideas. Other prefix codes (e.g., those in [Deo02, Fen96, Fen02, How97]) present other refinements with various tradeoffs. Theoretical exploration of the suite of algorithms from [Deo02] could illuminate other approaches than the ones we have taken.

Both our compression and indexing methods depend directly upon the space bounds of our dictionaries; any improvement there yields significant savings on our method. The best possible compression achievable is that empirically established by $E(L)$ in formula (3.2); however, as we saw in our experiments with Huffman encoding, RLE+γ encoding performs quite competitively with respect to Huffman codes in practice (and we didn't even count the space required for the prefix tree for Huffman encoding). Our key to space reduction is to exploit the underlying entropy in the text using a transform and a solid method of removing redundancy using the wavelet tree.

Chapter 4

Compressed Dictionaries and Data-Aware Measures

In this chapter, we propose measures for compressed data structures, in which space usage is measured in a data-aware manner. In particular, we consider the fundamental *dictionary problem* on *set data*, where the task is to construct a data structure for representing a set S of n items out of a universe $U = \{0, \ldots, u-1\}$ and supporting various queries on S. We use a well-known data-aware measure for set data called *gap* to bound the space of our data structures.

We describe a novel dictionary structure operating in near-optimal time that requires $\mathit{gap} + O(n \lg(u/n)/\lg n) + O(n \lg\lg(u/n))$ bits. Under the RAM model, our dictionary supports membership, rank, and predecessor queries in nearly optimal time, matching the time bound of Andersson and Thorup's predecessor structure [AT00], while simultaneously improving upon their space usage. We support select queries even faster in $O(\lg\lg n)$ time.

4.1 Introduction

The proliferation of data is a problem that is suffocating our abilities to manage information. Massive data sets from biological experiments, Internet routing information, sensor data, and audio/video devices require new methods for managing data. In many of these cases, the information content is relatively small compared to the size of the original data. We want to exploit the huge potential to save space in these cases. However, in many applications, data also needs to be indexed for fast query processing. The new trend of data structure design considers time and space efficiency together: The ultimate goal is to build structures that operate in the optimal (or nearly so) time bound, while requiring the minimum amount of space, tuned for the particular input data.

Ideally, the space required for a structure should be defined with respect to the *Kolmogorov complexity* of the data upon which the structure is built, as it is the space of the smallest program that can generate the input data. Unfortunately, it is undecidable for arbitrary input, making it an inconvenient measure for practical use. Thus, other measures of compressibility are used as a framework for data compression, like *entropy* for textual data.

One fundamental type of data is *set data*, which consist of a subset S of n items from a universe $U = \{0, \ldots, u-1\}$. Some specific examples include IP addresses, UPC barcodes, and ISBN numbers: set data also appear in inverted indexes for libraries and web pages, as well as results from scientific experiments. In many natural examples of set data, S is not a random subset of U and can be compressed. (For instance, consider a set S with a few tightly clustered items spread throughout U.)

In this chapter, we use the *gap* measure [BMNM$^+$93] (described formally in Section 4.2.2), which has been used extensively as a reasonable space measure in the context of inverted indexes [WMB99]. The gap measure counts the space required to encode the distances between successive items and is usually much less than the information-theoretic lower bound of $\lceil \lg \binom{u}{n} \rceil \approx n \lg(u/n)$ bits.[1] (This bound is known as the information-theoretic minimum because it is the minimum number of bits needed to differentiate the $\binom{u}{n}$ possible subsets of n items out of a universe of size u.) A *gap*-style encoding can be potentially much smaller than $\lceil \lg \binom{u}{n} \rceil$ bits for many of the data sets above, since it exploits short distances between items.

We use these notions of compressibility to design *compressed data structures* that index the data in a succinct way and also allow fast access. In particular, we address the fundamental *dictionary problem*, where we design a data structure to represent a subset S that supports various queries on S. In this chapter, we present compressed representations for both fully indexable dictionaries (FID) and indexable dictionaries (ID), improving the space required by previous results while maintaining near-optimal query time. In particular, un-

[1]Throughout the chapter, we assume the base of the logarithm is 2.

der the unit-cost RAM model, we develop a fully indexable dictionary (FID)—a data structure supporting rank and select queries—of size *gap* $+O(n \lg(u/n)/\lg n)$ $+O(n \lg\lg(u/n))$ bits, while supporting rank in time matching Andersson and Thorup's (nearly-optimal) predecessor structure [AT00] and select even faster in $O(\lg\lg n)$ time. When $n \in o(u)$, our fully indexable dictionary is asymptotically equal to *gap* space (with a constant of 1). This is important because, for most real-life data, $n \ll u$ and *gap* is significantly less than the worst-case information-theoretic minimum $\lceil \lg \binom{u}{n} \rceil$ bits. To our knowledge, this result is the first of its kind. Even when considered from a worst-case perspective, our data structures are the first to take $O(n \lg(u/n))$ bits with near-optimal query time. We also develop an indexable dictionary (ID)—a data structure supporting partial rank and select queries—in the same number of bits that supports each query even faster in $O(\lg\lg n)$ time. This result is the first to operate with gap-style bounds in space with time sublogarithmic in terms of the number of items stored. Moreover, our data structures are useful in practice; we also have a practical implementation and we discuss algorithmic engineering and experimental results in near the end of this chapter. Our results show that *gap* is about $10 - 40\%$ of $\lceil \lg \binom{u}{n} \rceil$ for many practical data sets.

The work in this chapter is a collaborative effort with Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter.

4.1.1 Comparisons to Previous Work

Previous results of Jacobson [Jac89b], Munro [Mun96], Brodnik et al. [BM99], Pagh [Pag99], and Raman et al. [RRR02] develop dictionaries that support constant-time queries. The best among these are the indexable dictionaries (ID) (supporting partial rank and select) and the fully indexable dictionaries (FID) (supporting rank and select) by [RRR02], both supporting constant-time queries. Their ID requires $\lceil \lg \binom{u}{n} \rceil + o(n) + O(\lg\lg u)$ bits, and their FID requires $\lceil \lg \binom{u}{n} \rceil + O(u \lg\lg u/\lg u) + O(\lg\lg u)$ bits. These results seem quite strong, as the constant factor associated with the information-theoretic minimum term is 1; unfortunately, the space is not bounded in a data-aware manner.

Recent work by Mäkinen and Navarro [MN06] and Sadakane and Grossi [SG06] achieves an FID with constant time queries taking $\mathit{gap} + O(n \lg\lg(u/n)) + O(u \lg\lg u/\lg u)$ bits of space.[2] Both of these data structures are meaningful as methods to achieve constant-time queries over a *gap* representation. Still, these FID structures do not work well when $n \ll u$, as the $o(u)$ term will be much (even exponentially) larger than the information-theoretic minimum term $\lceil \lg \binom{u}{n} \rceil$, dwarfing any savings we want to achieve. For instance, consider a typical example of maintaining a dictionary for IP lookup, storing say 2^{17} IP addresses out of a universe of size 2^{32}. In this case, $\lceil \lg \binom{u}{n} \rceil$ is roughly 345,661 (about 2^{18}) bits while their $o(u)$ term is roughly 6.71×10^8 (about 2^{29}) bits—several orders of magnitude larger than the information-theoretic minimum $\lceil \lg \binom{u}{n} \rceil$ bits.

Blandford and Blelloch [BB04] proposed an interesting scheme that allows easy transformation of any FID implemented with $O(n)$ pointers into another that requires $O(\mathit{gap}) + O(u^\alpha \lg u)$ bits for any $0 < \alpha < 1$.[3] After the transformation, query time is slowed down by a factor of $1/\alpha$ compared with time required by the original dictionary. Blandford and Blelloch's scheme allows us to have FIDs with space bounded in a data-aware manner. However, their analysis still has a potentially excessive $u^{\Omega(1)}$ term. We note that their method can be tuned by some of the techniques developed in this chapter to achieve $(1 + \epsilon)\mathit{gap}$ bits of space. However, this increases their search time by a multiplicative factor of $1/\epsilon$. In addition, they require either complex RAM operations or a decoding table that may require more space. This is in part because their space-savings approach is fundamentally different from our own; it packs a variable number of items into a constant number of memory words and fetches the information in a constant number of RAM operations or by use of a large decoding table. In contrast, our data structure fetches one item at a time. We describe this structure in more detail in Section 4.4.

A fundamental aspect of a dictionary's search capabilities is captured by the predecessor

[2]The middle term $O(n \lg\lg(u/n))$ comes from encoding the extra bits needed for a prefix code (such as a δ code).

[3]They only claim $O(n \lg((u + n)/n)) + O(u^\alpha \lg u)$ bits in their paper.

problem, since dictionaries that (implicitly) solve the predecessor problem require fundamentally more space and time than those that do not. Precisely, the predecessor query determines the largest item in S smaller than the query. Fredman and Willard [FW93] proposed the well-known fusion tree which supports predecessor queries in $O(\lg n/\lg\lg n)$ time. The query time was later improved by Beame and Fich's key result [BF99]. In particular, Beame and Fich describe a data structure taking $O(n^2 \lg u)$ bits of space that supports membership and predecessor queries in $BF(u,n) = O(\min\{(\lg\lg u)/(\lg\lg\lg u), \sqrt{(\lg n)/(\lg\lg n)}\})$ time. They also show that this bound is tight as long as we have only $O(n^{O(1)} \lg u)$ bits available.[4] Pǎtraşcu and Thorup [PT06] improved their space to $O(n^{1+\exp(-\lg^{1-\epsilon}\lg u)} \lg u)$ bits of space, but unfortunately this improvement does not help our data structure.

Andersson and Thorup [AT00] provide a transformation to Beame and Fich's data structure, improving the space to $O(n \lg u)$ bits and making the data structure dynamic using exponential search trees. However, the query time increases to

$$AT(u,n) = O\left(\min\left\{\sqrt{\frac{\lg n}{\lg\lg n}},\ \ \frac{\lg\lg u}{\lg\lg\lg u}\cdot\lg\lg n,\ \ \lg\lg n + \frac{\lg n}{\lg\lg u}\right\}\right).$$

Since rank and select can be used to answer predecessor queries, we improve Andersson and Thorup's structure in terms of space without sacrificing query time. In the worst case, our fully indexable dictionary compares favorably with both Raman et al. [RRR02] and Blandford and Blelloch [BB04]. With respect to the former, though we cannot support $O(1)$-time queries, we have eliminated the problematic $o(u)$ space term. Our query time—which is $AT(u,n)$—is already close to the optimal $BF(u,n)$. For our indexable dictionary, when compared with Raman et al.'s ID structure [RRR02], we pay a small price in the lookup time in exchange for achieving space bounds in terms of *gap*, which may be significant in practice.

The table in Figure 4.1 lists the theoretical results with practical estimates for the space required to represent the various compressed dictionaries we mentioned. In all reported

[4]It is this result that necessitates Raman et al.'s FID [RRR02] $o(u)$ space term, since constant-time rank and select queries imply constant-time predecessor queries as well.

bounds, we refer to *fully-indexable dictionaries* (FID). Note that $BF(u,n) \le AT(u,n)$ for any u and n.

Figure 4.1: Time and space bounds of dictionaries for *rank* and *select* queries.

	Theoretical		Practical[a]
Paper	Time	Space (bits)	Space (bits)
this chapter	$AT(u,n)$	$gap + o(\lg \binom{u}{n})$ when $n \ll u$	$\le 1,830,959$
[BB04]	$AT(u,n)$	$2gap + \Theta(u^\epsilon)$	$\le 1,855,116$
[vEBKZ77][b]	$O(\lg\lg u)$	$\Theta(n \lg u)$	$> 3,200,000$
[AT00]	$AT(u,n)$	$\Theta(n \lg u)$	$> 3,200,000$
[BF99]	$BF(u,n)$	$\Theta(n^2 \lg u)$	$> 320,000,000,000$
[PT06]	$BF(u,n)$	$\Theta(n^{1+\exp(-\lg^{1-\epsilon}\lg u)} \lg u)$	$> 10,000,000$
[Jac89b]	$O(1)$	$u + \Theta(u \lg\lg u / \lg u)$	$> 4,429,185,024$
[RRR02]	$O(1)$	$\lg \binom{u}{n} + \Theta(u \lg\lg u / \lg u)$	$> 136,217,728$
[MN06]	$O(1)$	$gap + O(n \lg\lg(u/n)) + \Theta(u \lg\lg u / \lg u)$	$> 136,017,728$
[SG06]	$O(1)$	$gap + O(n \lg\lg(u/n)) + \Theta(u \lg\lg u / \lg u)$	$> 136,017,728$

[a]The practical space bounds are for indexing our `upc_32` file, with n = 100,000 and $u = 2^{32}$. The values for [vEBKZ77, BF99, Jac89b, RRR02, MN06, SG06] are estimated by their reported space bounds. For these methods, we relaxed their query times to $O(\lg\lg u)$ to provide a fairer comparison in space usage.

[b]The theoretical space bound is from Willard's y-fast trie implementation [Wil84].

4.1.2 Outline of the Chapter

The organization of the chapter is as follows. In Section 4.2, we introduce three space measures for set data and show the strong relationship among them. In Section 4.3, we develop a binary searchable dictionary representation (`BSD`), which serves as an important component in our main results. In Section 4.4, we describe our fully indexable dictionary and analyze it for both gap-style bounds and worst-case bounds. We achieve a fully indexable dictionary supporting rank in $AT(u,n)$ time and select in $O(\lg\lg n)$ time, taking

$gap + o(n \lg(u/n))$ bits of space, or $O(n \lg(u/n))$ bits in the worst case. Note that fully indexable dictionaries that take $O(n^{O(1)} \lg u)$ bits of space are subject to the lower bound of [BF99]; hence, these times are near-optimal with respect to $BF(u,n)$. In Section 4.5, we present our indexable dictionary result, which cannot solve predecessor queries, and can thus improve upon the query times from [BF99]. Section 4.6 details our experimental findings. We conclude in Section 4.8.

4.2 Dictionaries and Data Aware Measures

Let $S = \langle s_1, \ldots, s_n \rangle$ be an ordered set of n items from a universe $U = \{0, 1, \ldots, u-1\}$ of size u; that is, $i < j$ implies $s_i < s_j$. We want to represent S in a succinct form so that we can perform basic dictionary queries on its compressed representation. We define dictionaries more formally in Section 4.2.1. The normal concern of a dictionary is how fast one can answer a query, but space usage is also an important consideration. We would like the dictionary to use the minimum space for representing S, regardless of how quickly it can be searched. There are some common measures to describe this minimum space. The first measure is $n \lg u$, which is the number of bits needed to store the items s_i explicitly in an array. The second measure is the information-theoretic minimum $\lceil \lg \binom{u}{n} \rceil \approx n \lg(u/n)$, which is the worst-case number of bits required to differentiate between any two distinct n-item subsets of universe U. In Section 4.2.2 we describe two more measures for representing the set S, motivating these as reasonable measures for analyzing the space required by a dictionary. We show strong relationships between these measures in Section 4.2.3, along with some experimental results that illustrate their relative performance.

4.2.1 The Dictionary Problem

The *dictionary* problem appears as a fundamental black box component in a number of applications used to offer fast access (for some queries, even constant-time access) to the data. Some examples include suffix arrays and IP lookup tries. Our interest is to exploit the great potential for a functional but compressed dictionary data structure. In some applications, dictionaries are the bottlenecks, both in terms of space and query time.

We describe some fundamental queries on set data. Here, $a \in U$. The $\mathit{member}(S, a)$ function indicates whether a appears in the set S. The $\mathit{rank}(S, a)$ function returns the number of items in S that are less than or equal to a. The $\mathit{select}(S, i)$ function returns the ith smallest item of S, for i ranging from 1 to n. The $\mathit{prank}(S, a)$ function is a rank function, but only for items of S. The $\mathit{pred}(S, a)$ function returns the predecessor of a, the largest item x in S such that $x < a$. We define these formally below.

$$\mathit{rank}(S, a) = \big|\{s_i | s_i \leq a\}\big|$$
$$\mathit{select}(S, i) = s_i$$
$$\mathit{member}(S, a) = 1 \text{ if } a \in S,\ 0 \text{ otherwise}$$
$$\mathit{prank}(S, a) = \mathit{rank}(S, a) \text{ if } a \in S,\ -1 \text{ otherwise}$$
$$\mathit{pred}(S, a) = \max\{s_i | s_i < a\} \text{ if } \mathit{rank}(S, a-1) > 0,\ -1 \text{ otherwise}$$

Jacobson [Jac89b] has discussed and motivated the power of *rank* and *select* functions at some length. In particular, he shows that the operation set $\{\mathit{rank}, \mathit{select}\}$ can perform more powerful queries than the operation set $\{\mathit{member}, \mathit{pred}\}$. As a result, much of the subsequent work has considered *rank* and *select* as fundamental operations on dictionary structures (such as [RRR02, Pag99, BB04]). To further illustrate this point, note that the right-hand column can be defined solely in terms of *rank* and *select*. For instance, $\mathit{member}(S, a) = \mathit{rank}(S, a) - \mathit{rank}(S, a-1)$ and $\mathit{pred}(S, a) = \mathit{select}(S, \mathit{rank}(S, a-1))$ if $\mathit{rank}(S, a-1) > 0$. We now define some convenient notation to describe different kinds of dictionaries.

Definition 4. An *indexable dictionary* (ID) represents a subset $S \subseteq U$ and supports the queries $\mathit{prank}(S, a)$ and $\mathit{select}(S, i)$. A *fully indexable dictionary* (FID) represents a subset $S \subseteq U$ and supports the queries $\mathit{rank}(S, a)$ and $\mathit{select}(S, i)$.

Fully indexable dictionaries can solve predecessor queries, and so they immediately find application in rich problem areas as IP lookup structures [CDG99], compressed text indexing [GGV03], and suffix arrays [GV00].

Suppose that for the set S of n items, each item s_i is also associated with a piece of satellite data d_i. To allow quick retrieval of the satellite data once the item is given, we could consider a set S' of tuples of the form $\langle \mathit{key}, \mathit{data} \rangle$, with $S' = \{\langle s_1, d_1 \rangle, \langle s_2, d_2 \rangle, \ldots, \langle s_n, d_n \rangle\}$, and build a dictionary on S'. In this context, we define $\mathit{lookup}(S', a) = d_j$ when $a = s_j$ for some j and `null` otherwise.

Definition 5. A *lookup dictionary* (LD) is a data structure representing a set S' that supports the query $\mathit{lookup}(S', a)$.

Let $A = d_1 d_2 \ldots d_n$ be a bitvector of length $|A| = \sum_i |d_i|$ with the data d_i concatenated together. If each piece of satellite data d_i is of a fixed length r, a simple array structure of $n \times r$ bits can be used to store the satellite data. We can construct an ID on S, so that for any item s_i, the *prank* query returns the position in A where its satellite data is stored. Combining this with RRR's ID result, we obtain the following lemma, which is used extensively in our data structures in Sections 4.4 and 4.5.

Lemma 28. *There exists a lookup dictionary (LD) with $m(q + r)$ bits supporting $\mathit{lookup}(S', a)$ in constant time, where $m = |S'|$, $q \leq \lg u$ is the number of bits to represent each key in S', and r is the number of bits for each satellite data.* □

When the satellite data are variable-length, we still store them using $\sum_i |d_i|$ bits. However, we need to know the starting position of each satellite data item. To do this, we store an ID on m items, where the ith item denotes the starting bit position of the ith piece of satellite data among the $\sum_i |d_i|$ possible positions. We ask *select* queries to determine the location of the ith satellite data item. The result of Blandford and Blelloch [BB05] on arrays of variable-length bitstrings also provides this functionality.

4.2.2 The *gap* and *trie* Measures

One well-known method for representing the set S is *gap encoding* [BMNM+93], which is often used in compressing inverted indexes. (We refer the reader to [WMB99] for a detailed treatment of the various applications of this method, as well as a source for further references.) Consider the gaps between consecutive items in S, where the ith gap g_i is equal to $s_i - s_{i-1}$. We can now represent the set S as the stream of gaps $G = g_1, \ldots, g_n$, where $g_1 = s_1$, along with the value n. The stream G of gaps can be stored using variable length encoding depending upon their size. Suppose we could store each g_i in $\lceil \lg(g_i + 1) \rceil$ bits. Then, the total space, which we call the *gap measure*, is

$$\mathit{gap}(S) = \sum_{i=1}^{n} \lceil \lg(g_i + 1) \rceil$$

bits. Note that we cannot merely store each g_i in $\lceil \lg(g_i + 1) \rceil$ bits and decode the stream uniquely; we also need to know the separation boundaries between successive items. One popular technique to "mark" these separations is by using a prefix code such as the δ code [Eli75]. In δ coding, we represent each g_i in $\lceil \lg(g_i + 1) \rceil + 2 \lceil \lg \lg(g_i + 1) \rceil$ bits, where the first $\lceil \lg \lg(g_i + 1) \rceil$ bits store the unary encoding of the number $\lceil \lg \lg(g_i + 1) \rceil$, the next $\lceil \lg \lg(g_i + 1) \rceil$ bits are the binary representation of the number $\lceil \lg(g_i+1) \rceil$, and the final $\lceil \lg(g_i+1) \rceil$ bits are the binary representation of g_i. We can then represent the stream of gaps $G = g_1, g_2, ..., g_n$ by concatenating the encoding of each g_i such that G is uniquely decodable. We refer to these extra bits of overhead beyond $\mathit{gap}(S)$ as the decoding overhead $Z(S)$. For δ coding, $Z(S) = 2 \sum_i \lceil \lg \lg(g_i + 1) \rceil$ bits. Our theoretical results in this chapter make use of the δ code.

Another example of a prefix code is the nibble code proposed in [BB04]. In this chapter, we will primarily use a variation of the nibble code called *nibble4* in our experiments. For this scheme, we write a "nibble" part of $\lceil \lceil \lg(g_i + 1) \rceil / 4 \rceil$ in unary, which is followed by $4 \cdot \lfloor \lceil \lg(g_i + 1) + 3 \rceil / 4 \rfloor$ bits to write the binary representation of g_i, padded out to multiples of four bits. (Later, we describe *nibble4fixed*, which we use for 64-bit data. It encodes the first part in binary in four bits, since for a universe size of 2^{64}, we would need to write $64/4 = 16$ different lengths.)

By Jensen's inequality,[5] $gap(S)$ is maximized when all gaps g_i are the same. In this case, $gap(S)$ would require roughly $n \lg(u/n)$ bits, since each of the n gaps would be of size u/n. $Z(S)$ is also maximized in this case for δ coding. Hence, $Z(S)$ is roughly $2n \lg \lg(u/n)$ bits. Other prefix codes, such as the γ code [Eli75] and some combination of Huffman and fixed-length coding, result in a somewhat different $Z(S)$. In this chapter, we use the δ encoding scheme and denote the bit representation of S using this encoding by $\mathtt{GAP}(S)$. The size of $\mathtt{GAP}(S)$ is $|\mathtt{GAP}(S)| = gap(S) + Z(S)$ bits.

Another method for compression of S is the *prefix omission method* (POM) [KS02], which is generally used to represent bitstrings of arbitrary length. Consider the bitstrings sorted lexicographically. We can represent each bitstring with respect to the previous bitstring by omitting the common prefix of the two. To compress S by POM, we think of each item of S as its $\lg u$-length bit representation. The POM for S can also be seen as a subtree (of n leaves) of the complete binary tree on u leaves (which is a trie). We denote this subtree by $Tree(S)$. Each left edge of $Tree(S)$ represents a $\mathbf{0}$, and each right edge represents a $\mathbf{1}$. Each root-to-leaf path in this trie defines an item s in S.

For $x, y \in S$, let $x \ominus y$ denote the bitstring formed by omitting the common prefix of x and y from the bit representation of x. More precisely, let $|lcp(x, y)|$ denote the length of the longest common prefix of x and y; then, $x \ominus y$ is the last $\lg u - |lcp(x, y)|$ bits of x. To represent S by POM, we generate the stream $L = l_1, l_2, \ldots, l_n$, where l_1 is the bit representation of s_1 in $\lg u$ bits and $l_i = s_i \ominus s_{i-1}$. Let $|l_i|$ denote the number of bits in l_i. Thus, the cost of this representation, which we call the *trie measure*, is

$$trie(S) = \sum_{i=1}^{n} |l_i| = |s_1| + \sum_{i=2}^{n} |s_i \ominus s_{i-1}|,$$

which equals the number of edges in $Tree(S)$. Similar to the gap measure, the above

[5]For a concave function f and $x_1 + x_2 + \cdots + x_k = x$, $\sum_i f(x_i)$ is maximized when $x_i = x/k$.

representation with *trie*(S) bits is not decodable as each string l_i is of variable length. Hence, we need some extra bits $Z'(S)$ for decoding, which takes $2\sum_i \lceil \lg |l_i| \rceil$ bits in the case of δ encoding. We use $\texttt{TRIE}(S)$ to denote the bit representation of S using POM, which takes $|\texttt{TRIE}(S)| = \mathit{trie}(S) + Z'(S)$ bits of space.

Let $S + a$ denote the set in which the positive integer a is added (modulo u) to each item of S. Thus, the set $S+a$ is $\{(s_1+a) \mod u, (s_2+a) \mod u, \ldots, (s_t+a) \mod u\}$. We define the *shifted trie measure* $\mathit{strie}(S) = \min_a\{\mathit{trie}(S+a)\}$, which corresponds to the number of bits needed to compress S by POM under the 'best shift'. We denote $\texttt{STRIE}(S)$ to be the corresponding $\texttt{TRIE}(S+a)$, and we define the space requirement $|\texttt{STRIE}(S)|$ similarly. Note that $|\texttt{STRIE}(S)|$ also includes the additional overhead of $\lg u$ bits to store the number a to retrieve the original S. Next, we argue that *trie*(S) could be somewhat larger than *gap*(S), but *strie*(S) is close to *gap*(S).

Below, we summarize the notation introduced in this section.

$\mathit{gap}(S) = \sum_{i=1}^{n} \lceil \lg(g_i+1) \rceil$	$\mathit{strie}(S) = \min_a\{\mathit{trie}(S+a)\}$
$\lvert\texttt{GAP}(s)\rvert = \mathit{gap}(S) + Z(S)$	$\lvert\texttt{STRIE}(s)\rvert = \mathit{strie}(S) + \lg u$
$\mathit{trie}(S) = \lvert s_1\rvert + \sum_{i=2}^{n} \lvert s_i \ominus s_{i-1}\rvert$	$\lvert x \ominus y\rvert = \lg u - \lvert \mathit{lcp}(x,y)\rvert$
$\lvert\texttt{TRIE}(s)\rvert = \mathit{trie}(S) + Z'(S)$	*Tree*(S) is a trie that stores the binary representations of items of S

4.2.3 Relationship Between *gap*, *trie* and *strie*

In this section, we show a strong relationship between the *gap*, *trie* and *strie* measures. For any item s_i, $\lceil \lg(g_i+1) \rceil$ is always smaller than $|l_i|$, but $|l_i|$ could be much larger. For example, when $s_{i-1} = 2^k - 1$ and $s_i = 2^k$, $|l_i| = k$ even though $\lceil \lg(g_i+1) \rceil = 1$. We show that this case cannot occur too frequently and prove that *trie*$(S) \leq 2$*gap*(S); furthermore, by applying a 'random shift', such cases are almost all eliminated. In the following lemma, we show that *trie*(S) can be more tightly bounded using this intuition.

Lemma 29. *The trie measure on the set $S+a$ requires $\mathit{trie}(S+a) \leq \mathit{gap}(S)+2n-2$ bits on average over all values of $a \in [1, u]$.*

Proof. We proceed by showing that the sum $\sum_a \mathit{trie}(S+a)$ is at most $u(\mathit{gap}(S) + 2n-2)$ bits. Recall that for a gap g_i, $|l_i|$ must be at least $\lceil \lg(g_i+1) \rceil$ bits long. For an arbitrary choice of a, $|l_i|$ can range from $\lceil \lg(g_i+1) \rceil$ to $\lg u$ bits in length. We count how many times each $|l_i|$ contributes to the sum. For an arbitrarily chosen gap g_i, there are exactly g_i values of a such that $|l_i|$ will branch from $\mathit{root}(\mathit{Tree}(S))$. Thus, the total cost incurred is $g_i \lg u$ bits. Similarly, there are $2g_i$ values of a such that $|l_i|$ would contribute $\lg u - 1$ bits to the sum. In general, for $j < \lg u - \lceil \lg(g_i+1) \rceil$, there are $2^j g_i$ values of a such that $|l_i|$ would contribute $\lg u - j$ bits to the sum. Finally, the number of times $|l_i| = \lceil \lg(g_j+1) \rceil$ is at most $u(2^{\lceil \lg(g_i+1) \rceil} - g_i)/2^{\lceil \lg(g_i+1) \rceil}$. Thus, $|l_i|$ contributes to the sum with

$$\sum_{j=0}^{\lg u - \lceil \lg(g_i+1) \rceil - 1} 2^j g_i(\lg u - j) + \frac{u\left(2^{\lceil \lg(g_i+1) \rceil} - g_i\right)}{2^{\lceil \lg(g_i+1) \rceil}} \lceil \lg(g_i+1) \rceil$$
$$= u\lceil \lg(g_i+1) \rceil - g_i \lg u + \frac{2ug_i}{2^{\lceil \lg(g_i+1) \rceil}} - 2g_i.$$

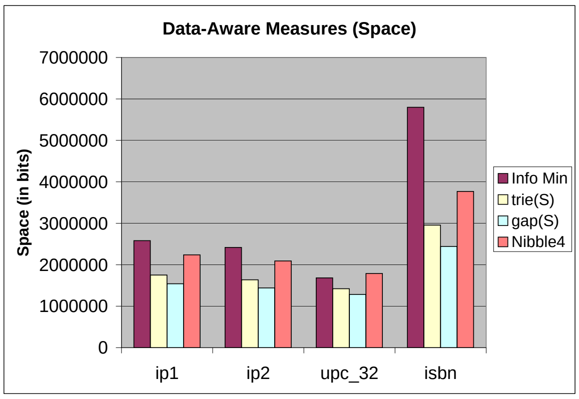
We also incur an additional cost associated with shifts such that $s_i + a > u$, where we charge $|l_i|$ with $\lg u$ bits, contributing an additional $g_i \lg u$ bits. Summing up and averaging over each of the u possible shifts, we see that the gap g_i requires an average of less than $\lceil \lg(g_i+1) \rceil + 2$ bits. We then sum this over all possible gaps, showing that an average $\mathit{trie}(S+a)$ is $\sum_{i=1}^{n}(\lceil \lg(g_i+1) \rceil + 2 - 2g_i/u) = \mathit{gap}(S) + 2n - 2$ bits, thus proving the lemma. □

Since the minimum is less than the average, we obtain the following corollary.

Corollary 5. *The shifted trie measure, $\mathit{strie}(S)$, is at most $\mathit{gap}(S) + 2n - 2$.*

Note that $|l_i|$ is bounded on average by $\lceil \lg(g_i + 1) \rceil + 2$ bits. Since the decoding overhead is $\lceil 2 \lg |l_i| \rceil$ with the δ code, we can bound the total overhead $2 \sum_i \lceil \lg |l_i| \rceil$ by $2n \lg \lg(u/n)$ bits using Jensen's inequality. Thus, the space requirement $|\texttt{STRIE}(S)|$ is at most $\mathit{strie}(S) + 2n \lg \lg(u/n) + \lg u$ bits.

We provide some experimental results on real data sets in Figure 4.2, which bear out the theoretical findings in this section. Here, the files tested are described in Section 4.6.1, and the space is reported (in bits) along the y-axis. The figure on the left shows data files with a universe of size $u \le 2^{32}$, and the figure on the right shows data files with $u \le 2^{64}$. Notice that $\mathit{gap}(S)$ is significantly smaller than $\lg \binom{u}{n}$ for real data. In fact, nibble4 is a decodeable gap encoding that *also* outperforms the information-theoretic minimum. Since $\mathit{gap}(S)$ is less than $\mathit{trie}(S)$ for all of the files, we are free to use the gap measure for the remainder of our experimental results.



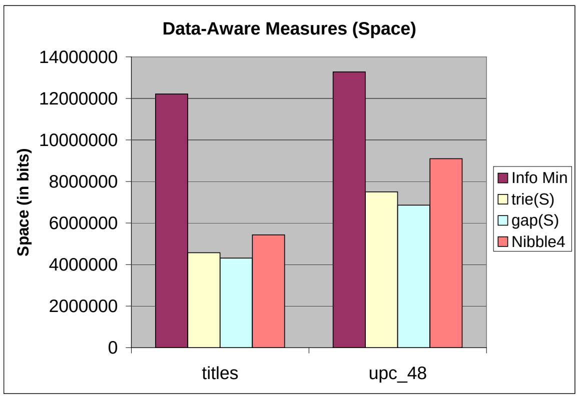


Figure 4.2: Comparison of $\lg \binom{u}{n}$, $\mathit{trie}(S)$, $\mathit{gap}(S)$, and a gap stream encoded according to the nibble4 code for the data files in Section 4.6.1.

4.3 Binary Searchable Dictionary Representation

Despite all the development on the POM model, the trie encoding of S does not support time-efficient queries as we would like. Klein and Shapira [KS02] use the trie encoding to search in compressed dictionaries, but their searching algorithm essentially consists of a linear scan of the items in the dictionary and takes at least $\Omega(n)$ time. Most algorithms using gap encoding also need a linear scan. In this section, we build a binary searchable data structure BSD, which resolves rank and select queries in $O(\lg n)$ time. We show that the space required by this structure is *gap* bits plus low-order terms. In fact, the main point of this section is in showing that a binary-searchable representation requires about the same number of bits as simple linear encoding schemes. Also, BSD is our main building block and will be used later in this chapter to support fast lookup in our FID and ID dictionary structures.

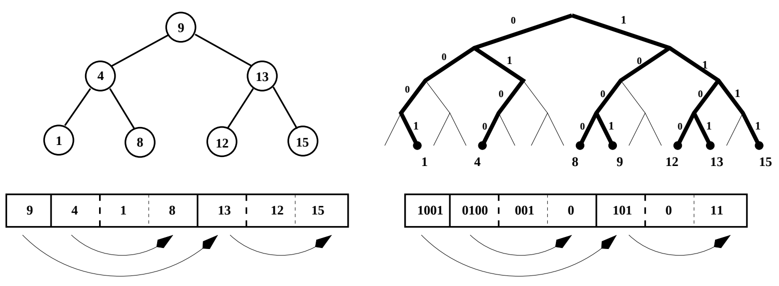


Fig. A. The left hand side shows a binary search tree built on the items 1, 4, 8, 9, 12, 13, and 15. Beneath that is its pre-order layout on disk, where the arrows represent pointers to the right subtree. The right hand side shows the trie built on the same items. Beneath that is the corresponding layout on disk, but each item s is encoded with respect to $anc(s)$. For instance, 8 is encoded in the layout on the right as **0**, since $anc(8) = 9$ differs from it by a single bit.

The BSD structure encodes a pre-order traversal of a balanced binary search tree T built on the n items of S. In Figure A, the pre-order traversal for the set S is 9, 4, 1, 8, 13, 12, and 15. The key point is that instead of storing each item s_i explicitly in $\lg u$ bits, we encode an item with respect to an ancestor $anc(s_i)$, defined as follows. Let A_i be the set of all the ancestors of s_i in the binary search tree T. Then,

$anc(s_i) = x \in A_i$ such that $lcp(s_i, x)$ is maximized over all ancestors in A_i. We represent s_i by $s_i \ominus anc(s_i)$ using $\lg u - |lcp(s_i, anc(s_i))|$ bits, reminiscent of our trie encoding. Now we define the $\texttt{BSD}(S)$ encoding.

We use a recursive layout to describe the pre-order traversal of the binary search tree of n items. Let the subsets $S_L = \langle s_1, s_2, ..., s_{\lceil n/2 \rceil - 1} \rangle$ and $S_R = \langle s_{\lceil n/2 \rceil + 1}, ..., s_n \rangle$ represent the left and right subtrees of the $s_{\lceil n/2 \rceil}$th item. Generally, let $S_{i,j} = \langle s_i, s_{i+1}, ..., s_j \rangle$. Let $anc(s_{\lceil n/2 \rceil}) = 0$. For $\texttt{BSD}(S)$, let $|\texttt{BSD}(S)|$ denote the number of bits needed to encode $\texttt{BSD}(S)$. Then, we define the $\texttt{BSD}$ encoding as

$$\texttt{BSD}(S) = \langle s_{\lceil n/2 \rceil} \ominus anc(s_{\lceil n/2 \rceil}); |\texttt{BSD}(S_L)|; \texttt{BSD}(S_L); \texttt{BSD}(S_R) \rangle.$$

Note that $s_{\lceil n/2 \rceil} \ominus anc(s_{\lceil n/2 \rceil})$ is a variable-length string, which is stored using δ coding. The term $|\texttt{BSD}(S_L)|$ constitutes additional overhead but is needed in order to jump to the right half of the set while searching. (We will call this term the *pointer cost*, and we will refer to it in our experimental section.) In fact, we could actually store just $\min\{|\texttt{BSD}(S_L)|, |\texttt{BSD}(S_R)|\}$ bits (with an additional n bits to indicate our choice), along with remembering whichever was smaller of the left and right subtrees.[6] Nevertheless, it turns out that $\texttt{BSD}$ requires nearly the same space as does the $\texttt{TRIE}$ encoding. Next, we describe how rank and select functions can be supported in $O(\lg n)$ time using $\texttt{BSD}(S)$, and then we analyze the space usage of $\texttt{BSD}(S)$.

We use $\texttt{BSD}(S)$ as a black box on $O(\lg n)$ items and achieve $O(\lg \lg n)$ time; however, in order to do so, we must be able to decode a δ-coded item (or bitstring) in $O(1)$ time in the RAM model. We assume that the word size of the machine is at least $\lg u$ bits, and that we are allowed to perform addition, subtraction, multiplication, and bitshift operations in $O(1)$ time. We also assume that we can calculate the position of the leftmost $\mathbf{1}$ of a subword x of $\lg \lg u$ bits in $O(1)$ time. (This task is

[6]Making this improvement would require the structure to be built from the bottom-up rather than with our recursive formulation above; we defer those details in the interest of clarity.

equivalent to calculating $\lceil \lg(x+1) \rceil$ when the word x is seen as an integer.) We can also easily encode and decode the $\ominus$ operator using bitshifts and additions. These assumptions are sufficient to allow $O(1)$ decoding time. If this model is not applicable, we can simulate the decoding by explicitly storing the decoding result of every possible $\lg \lg u$-bit number in a table with $\lg u$ entries. Note that this table takes $O(\lg u \lg \lg \lg u)$ bits, which is negligible overhead.[7]

In order to support *rank* and *select*, we just need to store the single value n (in $\lg n$ bits) at the beginning of the `BSD` to indicate how many items are stored within the structure. Since our structure is a well-defined balanced binary tree, at any node x with n_x items, we know that the size of our left subtree contains $\lceil n_x/2 \rceil - 1$ items, and our right subtree contains $n_x - \lceil n_x/2 \rceil$ items. Hence, we can compute *rank* and *select* based upon this information. More precisely, given $\texttt{BSD}(S)$, $\mathit{rank}(S, a)$ and $\mathit{select}(S, i)$ can be computed in $O(\lg n)$ time by calling the recursive functions $\mathit{rrank}(\texttt{BSD}(S), a, 0, u, n)$ and $\mathit{rselect}(\texttt{BSD}(S), i, 0, u, n)$ as detailed below. In the pseudocode, the function $\mathit{root}(B)$ returns the first encoded string in B (i.e., $\mathit{root}(B) = s_i \ominus \mathit{anc}(s_i)$), and the function $\mathit{decode}(x, \ell, r)$ returns the item s_i that corresponds to the root of B. The latter function can be computed by first determining $\mathit{anc}(s_i)$, which is one of ℓ or r based on the first bit of $\mathit{root}(B)$. Then,

[7]We could reduce the size of this table even further to $O(\lg \lg n \lg \lg \lg \lg n)$ bits by using a slightly different encoding scheme than the δ code.

$s_i = (anc(s_i) \text{ div } 2^y) \times 2^y + root(B)$, where $y = |root(B)|$.

```
function $rrank(B, a, \ell, r, n)$ {
    if $(n = 0)$ return $0$;
    $x \leftarrow root(B)$;
    $z \leftarrow decode(x, \ell, r)$;
    if $(z = a)$ return $\lceil n/2 \rceil + 1$;
    else if $(z < a)$
     return $\lceil n/2 \rceil +$
       $rrank(\texttt{BSD}(S_R), a, z, r, n - \lceil n/2 \rceil)$;
    else return $rrank(\texttt{BSD}(S_L), a, \ell, z, \lceil n/2 \rceil - 1)$;
}
```

```
function $rselect(B, i, \ell, r, n)$ {
  $x \leftarrow root(\texttt{BSD}(S))$;
  $z \leftarrow decode(x, \ell, r)$;
  if $(i = \lceil n/2 \rceil)$ return $z$;
  else if $(i > \lceil n/2 \rceil)$
   return
    $rselect(\texttt{BSD}(S_R), i - \lceil n/2 \rceil, z, r, n - \lceil n/2 \rceil)$;
  else return
   $rselect(\texttt{BSD}(S_L), i, \ell, z, \lceil n/2 \rceil - 1)$;
}
```

We denote the $rank(S, a)$ and $select(S, i)$ that operate on $\texttt{BSD}(S)$ by the functions $BSD\_rank(B, a)$ and $BSD\_select(B, i)$, where B is a pointer (of $\lg u$ bits) to $\texttt{BSD}(S)$.

Lemma 30. *The* $\texttt{BSD}(S)$ *representation requires at most* $trie(S) + O(n \lg\lg(u/n))$ *bits and supports rank and select functions in* $O(\lg n)$ *time.*

Proof. The space of $\texttt{BSD}(S)$ can be divided into three parts: (i) the space for all $s_i \ominus anc(s_i)$; (ii) their decoding overhead; and (iii) the space to encode all $|\texttt{BSD}(S_L)|$, used to jump to the right half of the encoding. We now describe the space required for each of these parts.

The space for (i) can be shown to be equal to the number of edges in *Tree*(S), which is exactly *trie*(S). To prove this, it suffices to show that each edge in *Tree*(S) is encoded only once in its $\texttt{BSD}(S)$ representation. Let item s be encountered according to its pre-order binary search tree traversal. Let A be the set of all *ancestors* on the root-to-leaf path leading to s in the binary search tree. In the trie structure, the path to s must lay between two root-to-leaf paths in the trie: either the path leading to its rightmost encoded ancestor on its left l or its leftmost encoded ancestor on its right r. We encode $s \ominus anc(s)$, which must either be l or r. (This could be the parent of s.) Since no other edge in the trie that lies between the path to l and the path to r has been used thusfar, each trie edge is encoded only once in any `BSD` structure.

For (ii), the overhead is analogous to $Z(S)$ and we can bound it by $O(n \lg\lg(u/n))$ using Jensen's inequality. In particular, we must encode the length of the new branch for s. Essentially, we are encoding n items out of a universe of *trie* bits to indicate the starting bit position of each branch's encoding. By Jensen's inequality, the worst case for this encoding occurs when all n items encode the length trie/n, requiring at most $n \lg(\mathit{trie}/n) \leq n \lg((2 \sum_i \lg g_i)/n) = O(n \lg\lg(u/n))$ bits. We must also know $\mathit{anc}(s)$, the ancestor we chose to encode from. We remember our choice automatically according to the first bit of the encoded string—a leading bit of **0** means we chose r and a leading bit of **1** means we chose l.

For (iii), we analyze this by considering the contribution of $|\texttt{BSD}(S_L)|$ at each level of the binary search tree of S. At level 1, i.e. the root level, $|\texttt{BSD}(S_L)|$ is at most $\lg(n \lg(u/n))$ bits. At level i, this contribution is maximized (by Jensen's inequality) when all of the 2^{i-1} contributing terms are equal. (In other words, all trees are the same size.) Thus, the space usage at level i is bounded by $2^{i-1} \lg((n/2^{i-1}) \lg(u/n))$. Summing up, we have

$$\sum_{i=1}^{\lg n} 2^{i-1} \lg\left(\frac{n}{2^{i-1}} \lg \frac{u}{n}\right) = O\left(n \lg\lg \frac{u}{n} + n\right),$$

which is a path recursion sum [GK81]. □

The above lemma suggests that $\texttt{BSD}(S+a)$ would require fewer than than $\mathit{trie}(S+a)$ bits, plus $O(n \lg\lg(u/n))$ bits for any a. Thus by Corollary 5, $\min_a\{|\texttt{BSD}(S+a)|\}$ is at most $\mathit{gap}(S) + O(n \lg\lg(u/n))$ bits. For the rest of the chapter, we assume the `BSD` representation for S is based on its best possible shift. Thus, we obtain the following theorem, which will be used in further construction of our data structures in Sections 4.4 and 4.5.

Theorem 17 (BSD). *The representation* `BSD`(S) *is a fully indexable dictionary (FID) occupying* $gap(S) + O(n \lg\lg(u/n))$ *bits, while supporting rank and select functions in* $O(\lg n)$ *time.* □

Next, we describe `BSGAP`(S), a simple and implementable variant of the `BSD`(S) representation that we use in our experimental results in Section 4.6. The key idea of `BSGAP`(S) is to directly encode the difference $|s_i - anc(s_i)|$ using gap encoding. Precisely, we replace the encoding $s_i \ominus anc(s_i)$ from `BSD`(S) by $\lceil \lg(|s_i - anc(s_i)| + 1) \rceil$. We also store one additional bit to indicate which ancestor encodes s_i. Using a similar analysis to that in Lemma 30, we arrive at the following corollary.

Corollary 6. *The representation* `BSGAP`(S) *is a fully indexable dictionary (FID) occupying* $gap(S) + O(n \lg\lg(u/n))$ *bits while supporting rank and select functions in* $O(\lg n)$ *time.* □

Proof. It suffices to show that for each item s_i, its encoding in `BSGAP` is no more than in `BSD`. Let $g_i = |s_i - anc(s_i)|$ be the gap we wish to encode, and let $|l_i| = \lg u - |lcp(s_i, anc(s_i))|$ be the length of its encoding in `BSD`. Recall that for any gap g_i, $|l_i|$ must be at least $\lceil \lg(g_i + 1) \rceil$ bits long. Thus, under a random shift, $|l_i|$ can range from $\lceil \lg(g_i + 1) \rceil$ to $\lg u$ bits in length. Since `BSGAP` encodes s_i in the minimum required, we automatically arrive at the first space term $gap(S)$. The `BSGAP` representation also requires an additional n bits to indicate which ancestor (of l or r) encodes the current gap; this is accounted for in the second space term. The rest of the `BSGAP` representation follows from `BSD`. □

Though the `BSGAP` data structure seems to do little more than avoid an arbitrary shift, its consequences are far more interesting: `BSGAP` illustrates that a non-consecutive gap structure can still achieve *gap*-style bounds. In a sense, `BSGAP` presents a way to store each of the n nodes in a binary search tree for S in fewer than $\lg u$ bits. Moreover, it's an extremely simple (and implementable) technique.

4.4 The Fully Indexable Dictionary Structure

In this section, we describe our first main result, Theorem 18. We build a simple two-level hierarchical framework to obtain a fully indexable dictionary (FID) such that *rank* takes $AT(u, n)$ time and *select* takes $O(\lg\lg n)$ time. The challenge in designing such a data structure lies in only spending $gap(S) + O(n \lg(u/n)/\lg n) + O(n \lg\lg(u/n))$ bits in the process.

We describe our structure in a bottom-up way. At the bottom level, we store a BSD dictionary for every $\lceil \lg^2 n \rceil$ items from set S, each of which can resolve a rank or select query in $O(\lg\lg n)$ time. We also store $B.first\_rank$ along with each BSD B, where $B.first\_rank$ is the rank in S of its first item in B. We also keep an array $P[1..\lceil n/\lg^2 n \rceil]$, where $P[i]$ stores a pointer to the ith BSD structure, which stores the items $s_{(i-1)\lg^2 n+1}, \ldots, s_{i \lg^2 n}$. This structure alone is sufficient to support *select*. In order to support *rank*, let $\hat{S} = \{s_i | i \mod (\lg^2 n) = 1\}$ be the set of smallest items from each BSD. We build an instance of Andersson and Thorup's predecessor structure [AT00] on $\hat{S}$, called R. To support *rank*, we use a lookup dictionary L from Lemma 28 built on $\hat{S}$ as keys with pointers to the corresponding BSD as satellite data. We denote the process of looking up the satellite data associated with $s \in \hat{S}$ by $L.lookup(s)$. Then, *rank* and *select* can be solved as follows.

```
function $rank(S, a)$ {
    $s \leftarrow pred(R, a)$;
    $B \leftarrow L.lookup(s)$;
    return
      $B.first\_rank + BSD\_rank(B, a)$;
}
```

```
function $select(S, i)$ {
    $j \leftarrow \lceil i/(\lg^2 n) \rceil$;
    $B \leftarrow P[j]$;
    return
      $BSD\_select(B, i - B.first\_rank + 1)$;
}
```

We are almost ready to show the main theorem of this section, but first, we require the following lemma.

Lemma 31. *Let $S_1, S_2, ..., S_k$ be a partition of S, with each S_i consisting of items of consecutive ranks in S. Precisely, each S_i consists of items $s_j, s_{j+1}, ..., s_\ell$ for some $j \leq \ell$. Then, $\sum_{i=1}^{k} |\texttt{BSD}(S_i)| \leq gap(S) + O(k \lg u) + O(n \lg\lg(u/n))$.*

Proof. Let $u_i = \max\{s \in S_i\} - \min\{s \in S_i\} + 1$ and $n_i = |S_i|$. By Theorem 17, $|\texttt{BSD}(S_i)| \leq gap(S_i) + O(n_i \lg\lg(u_i/n_i))$. Thus, the lemma follows since $\sum_{i=1}^{k} gap(S_i) \leq gap(S) + O(k \lg u)$, and by Jensen's inequality, we show that $\sum_{i=1}^{k} O(n_i \lg\lg(u_i/n_i)) \leq O(n \lg\lg(u/n))$. □

Based on the above lemma, we obtain the main theorem below, along with a worst-case analysis in Corollary 7, since *gap* and $O(n \lg\lg(u/n))$ are bounded by $O(n \lg(u/n))$.

Theorem 18. *We implement a fully indexable dictionary (FID) using a total of $gap(S) + O(n \lg(u/n)/\lg n) + O(n \lg\lg(u/n))$ bits so that rank queries take $AT(u, n)$ time and select queries take $O(\lg\lg n)$ time.*

Proof. For *select*, we require $O(\lg\lg n)$ time to traverse the ith `BSD` dictionary. For *rank*, the time bound is dominated by the predecessor query in R, taking $AT(u, n/\lg^2 n) = O(AT(u, n))$ time. This shows our time bounds. For our space bounds, the $n/\lg^2 n$ `BSD` structures require a total of $gap(S) + O(n \lg(u/n)/\lg n) + O(n \lg\lg(u/n))$ bits. The array P and the field $B.first\_rank$ take at most $O(n/\lg^2 n) \times \lg u = O(n \lg(u/n)/\lg n)$ bits in total, proving the theorem. □

Corollary 7. *We implement a fully indexable dictionary (FID) using no more than $O(n \lg(u/n))$ bits so that rank queries take $AT(u, n)$ time and select queries take $O(\lg\lg n)$ time.* □

Finally, we capture a technically interesting space-time tradeoff of our FID, obtained by scaling the size of the groupings. This observation implies that the second-order space term in our structure can be made arbitrarily small, at the cost of a slight increase in the query times.

Corollary 8. *For any $\alpha > 1$, we can implement a fully indexable dictionary (FID) in total space $gap(S) + O(n \lg(u/n)/\lg^{\alpha-1} n) + O(n \lg\lg(u/n))$ bits so that the function rank takes $AT(u, n/\lg^{\alpha} n) + O(\alpha \lg\lg n)$ time and the function select takes $O(\alpha \lg\lg n)$ time.* □

4.5 The Indexable Dictionary Structure

In this section, we build upon the approach of the last section. We partition S into lower level `BSD` structures, each of size *at most* $\lg^3 n$. We use a top level 'distributor' structure which enables us to access the correct `BSD` while answering a query. In contrast to the last section, if the query item is not present in S, our top level distributor may not return any associated `BSD`. Hence, we cannot support rank or predecessor queries.

Our top level distributor takes $O(\lg\lg n)$ time to return the correct `BSD`. This is less than $AT(u, n)$ time; the partitioning scheme is somewhat more complex than that in our FID. As a result, we can support partial rank or select queries in $O(\lg\lg n)$ time. To manage the space required, we limit the number of partitions to be at most $O(n \lg\lg n/\lg^3 n)$, so that the overhead incurred by our top level distributor can be bounded by the same second-order term as in our FID.

Next, we describe our top level distributor structure, which is analogous to the van Emde Boas (VEB) tree [vEBKZ77]. With this distributor structure, on any given input x, we can report x is not in S, or obtain the `BSD` that can contain x efficiently.

4.5.1 The Top Level Distributor Structure

Our distributor structure is a recursive structure analogous to a VEB tree. Instead of having $O(\lg\lg u)$ levels of recursion as in the case for a VEB tree, our distributor has only $h = 3\lg\lg n$ levels. At the top level (Level 1), we have a single distributor (with parameter $p = 0$ to be explained shortly) to distribute all items in S. For level $i = 1$ to $h-1$, a Level i distributor with parameter p connects to some Level $i+1$ distributors, which are then used to distribute the items recursively; the parameter p indicates that all the input items share the same first p bits. At the bottom level (Level h), a Level h distributor directs the items to their designated `BSD` structures. More precisely, for $i = 1$ to $h-1$, a Level i distributor with parameter $p = p_i$ works as follows:

1. Partition the items into groups according to the first $p_i + (\lg u)/2^i$ bits.
2. For each group with more than $\lg^3 n$ items (which we call a dense group), the items are passed to a Level $i+1$ distributor with parameter $p = p_i + (\lg u)/2^i$.
3. For all items not in a dense group, they are grouped together.
   (a) If the number of items is at most $\lg^3 n$, the items are passed to a Level h distributor with parameter $p = p_i$.
   (b) Otherwise, the items are passed to a Level $i+1$ distributor with parameter $p = p_i$.

We can easily show the following by the recursive definition above: At a Level i distributor with parameter $p = p_i$, if we partition the items into groups based on the first $p_i + (2\lg u)/2^i$ bits instead, the size of each group is at most $\lg^3 n$. Making use of this fact, a Level h distributor with parameter $p = p_h$ partitions the n_h input items into groups based on their first $p_h + (2\lg u)/2^h$ bits, such that each group is of size at most $\lg^3 n$. The n_h items are then directed to the designated `BSD` data structures, with each `BSD` containing at most $O(\lg^3 n)$ items. With the above data structure D, we can find the `BSD` that can contain x by calling $find\_BSD(D, x)$ as follows:

```
function $find\_BSD(D, x)$ {
  $D_1 \leftarrow$ Level 1 distributor from $D$;
  $i \leftarrow 1$, $p_1 \leftarrow 0$;
  for $i = 1$ to $h - 1$
    $(D_{i+1}, p_{i+1}) \leftarrow distribute(D_i, i, p_i, x)$;
    if ($D_i$ is a Level $h$ distributor)
      $p_h \leftarrow p_i$, break;
  return $retrieve\_BSD(D_h, p_h, x)$;
}
```

```
function $retrieve\_BSD(D, p, x)$ {
  $L \leftarrow$ the LD stored in $D$;
  $y \leftarrow x[p + 1..p + (2\lg u)/\lg^3 n]$;
  return $L.lookup(y)$;
}
```

The function $distribute(D_i, i, p_i, x)$ retrieves the Level $i+1$ distributor with parameter $p = p_i$ in which x is distributed according to the first $p_i + (\lg u)/2^i$ bits. The notation $x[\ell..r]$ ($\ell \leq r$) denotes the substring of the bitstring representation of x, starting at the ℓth bit and ending at the rth bit. The function $L.lookup(y)$ returns $lookup(S(L), y)$ if $y \in S(L)$, where $S(L)$ denotes the set of keys stored by L.

Once we obtain the `BSD` B that can contain x, determining whether x is in B can be done in $O(\lg\lg n)$ time. Thus, if $find\_BSD(D, x)$ can be done in $O(\lg\lg n)$ time, the total time to answer $member(S, x)$ is also $O(\lg\lg n)$.

4.5.2 Distributor Details

In this part, we give details of the distributor that supports $distribute(D_i, i, p_i, x)$ at Level i ($i \in [1, h-1]$) and $retrieve\_BSD(D_h, p_h, x)$ at Level h efficiently. We make use of an LD of Lemma 28 to achieve this. Based on this implementation, we show that $find\_BSD(D, x)$ can be done in $O(\lg\lg n)$ time.

For $i = 1$ to $h - 1$, a Level i distributor with parameter p maintains an LD of Lemma 28 that stores the $p + (\lg u)/2^i$ bits corresponding to a dense group as keys, and storing the $\lg u$-bit pointer to the corresponding Level $i+1$ distributor as satellite information. It also explicitly stores an 'escape' pointer to the Level h or the Level $i+1$ distributor that corresponds to items not in dense groups.

For a Level h distributor with parameter p, we use a different structure. Let n_h be the number of items managed by this distributor. We store the number k of distinct `BSD`s containing these n_h items and an array $A[1..k]$ storing the pointers to these `BSD`s. Recall that all the n_h items share the first p bits, and the distributor here distributes an item into a group according to the first $p + (2\lg u)/2^h$ bits. Therefore, we maintain an LD of Lemma 28 for the $(2\lg u)/2^h$ bits that corresponds to a non-empty group, starting at the $(p+1)$st position. For the satellite information, we store the array entry of the corresponding `BSD`, which again takes $(2\lg u)/2^h$ bits.

A Minor Modification. If each `BSD` data structure corresponds to items in consecutive ranks, we can bound the total space by $\mathit{gap} + O(n \lg\lg(u/n))$ bits. Unfortunately, in the current scheme, a `BSD` data structure directed by a Level h distributor may not correspond to items of consecutive ranks. For instance, let s_i and s_j be two items in the same `BSD`; then at some level, an intermediate item s_{i+1} may be partitioned into a dense group, while s_i and s_j are items not in the dense group. Consequently, the intermediate item s_{i+1} is not stored in the same `BSD` as s_i and s_j.

In order to bound the space as desired, we use a little fix: for each existing `BSD` in the current scheme, we split the items into maximal groups of consecutive ranks, and store each group in a separate `BSD`. Essentially, we transform the existing `BSD` into a list of `BSD`s so that each new `BSD` corresponds to items of consecutive ranks. Then, a Level h distributor now directs the item into one of the k lists of `BSD`s (as opposed one of the k `BSD`s before). We store an array $A[1..k]$ for the pointers to the k lists; for each list, we store the number k' of `BSD`s it contains. (Note that $k' \leq \lg^3 n$, since the total number of items in a BSD is $O(\lg^3 n)$.) We also store an array $B[1..k']$ such that $B[i]$ stores the pointer to the BSD whose smallest item is the ith smallest among that of the other BSDs. With the above implementation, $distribute(D, i, p, x)$ (Lines 3 and 6 in $find\_BSD(D, x)$) for each $i = 1$ to $h - 1$ can be done in $O(1)$ time. Then at Level h, we obtain the list of BSDs that can contain x in $O(1)$ time. After that, we use binary search on x against the smallest items of the BSDs to find the BSD that can contain x (Line 8). The time required is $O(\lg k')$, which is at most $O(\lg \lg n)$ since $k' \leq \lg^3 n$. Then, $find\_BSD(D, x)$ can be done in $O(\lg \lg n)$ time.

4.5.3 Solving Partial Rank and Select Queries

The partial rank query can be readily supported by our data structure in $O(\lg \lg n)$ time, as shown in the pseudo-code below. To enable the select query, we additionally maintain an array $F[1..n/\lg^3 n]$ such that $F[i]$ stores a pointer to a list of BSDs that can contain the items with rank in $[(i-1)\lg^3 n, i \lg^3 n]$. For each list, we store number k'' of BSDs in the list, and an array $G[1..k'']$ for pointers to the k'' BSDs such that $G[1].first\_rank < G[2].first\_rank < G[3].first\_rank < ... < G[k''].first\_rank$.

Then, *select*(S, j) can be solved in the following pseudo-code.

```
function prank(S, x) {
  $B \leftarrow find\_BSD(D, x)$;
  if ($B =$ null) return -1;
  else
    $r \leftarrow B.first\_rank$;
    $r' \leftarrow BSD\_rank(B, x)$;
    if ( $BSD\_select(r', B) = x$ )
       return $r + r' - 1$;
    else return -1;
}
```

```
function select(S, j) {
  $G \leftarrow F[\lceil j/\lg^3 n \rceil]$;
  $k'' \leftarrow$ the number of BSDs in the list $G$;
  $i \leftarrow BinarySearch(G, k'', j)$;
  $r_i \leftarrow G[i].first\_rank$;
  return $BSD\_select(j - r_i + 1, G[i])$;
}
```

The function $BinarySearch(G, k'', j)$ returns i such that $G[i].first\_rank < j < G[i+1].first\_rank$ using binary search, which takes $O(\lg k'')$ time. The total time required for *select* is $O(\lg k'') + O(\lg\lg n) = O(\lg\lg n)$.

4.5.4 Space Analysis

To bound the total space usage, we will make use of the following lemma.

Lemma 32. *We show that*

1. *the total number of distributors, $\sum_{i=1}^{h} d_i$, is at most $O(n \lg\lg n/\lg^3 n)$, and*

2. *the total number of* BSD *data structures is at most $O(n \lg\lg n/\lg^3 n)$.*

Proof. For all the distributors in our data structure, we use $Dist(r, p, i)$ to denote the Level i distributor such that all the items managed by it share the same prefix r of length p. We call a distributor *dense* if it manages more than $\lg^3 n$ items; otherwise, it is called *sparse*. Note that sparse distributors only occur at Level h.

For Level i, the number of dense distributors is at most $n/\lg^3 n$, because the items they manage are disjoint. Thus, there are at most $3n \lg\lg n/\lg^3 n$ dense distributors in total. For each sparse distributor $Dist(r, p, h)$, there must exist a dense distributor $Dist(r, p, i)$ for some i. We map $Dist(r, p, h)$ to $Dist(r, p, i)$ such that i is maximized. Note that it is a bijection. Thus, the number of sparse distributors is bounded by the number of dense distributors, and the first claim follows.

If two consecutive rank items s_j and s_{j+1} are stored in different `BSD`s, we call (s_j, s_{j+1}) a *cut*. A cut can happen in one of two ways: (1) if s_j and s_{j+1} come from two distributors, or (2) if s_j and s_{j+1} come from the same Level h distributor which is dense. Note that the number of cuts is equal to the number of `BSD`s. Now, we count the number of cuts as follows.

For cuts of the first type, consider the smallest level i such that the s_j and s_{j+1} are in different distributors, say D_ℓ and D_r. (This implies that they are at the same Level $i-1$ distributor.) Then, by the definition of a distributor, either D_ℓ or D_r must be dense. We associate the cut with the dense distributor(s). Then, in this mapping, a dense distributor can be associated with at most two cuts, namely when it takes the roles of D_ℓ and D_r, respectively. Thus, the number of cuts of the first type is bounded by the number of dense distributors, which is $O(n \lg\lg n/\lg^3 n)$.

The number of cuts of the second type is, by definition, bounded by $O(n/\lg^3 n)$. Thus, the second claim follows. □

Next, we notice that for a particular i, items managed by different Level i distributors are disjoint. Let d_i denote the number of Level i distributors in our data structure. Also, recall that the space for an LD is $O(m(q+r))$ bits where m is the number of items, q is the number of bits needed to represent each key (i.e., p_i bits for the LD in a Level i distributor, and $2\lg u/\lg^3 n$ bits for the LD in a Level h distributor), and r is the number of bits for each satellite data (i.e., $\lg u$ bits for the LD in a Level i

distributor, and $2\lg u/\lg^3 n$ bits for the LD in a Level h distributor). Then, for any i in $[1, h-1]$, the space occupied by all Level i distributors is equal to the space of LD for dense groups + space for escape pointers $\leq O(n/\lg^3 n \times \lg u) + d_i \lg u$ bits. On the other hand, the space occupied by all Level h distributors is equal to space of LD for non-empty groups + space for k + space for arrays $A[1..k]$ and k' + space for arrays $B[1..k'] \leq O(n \times 2\lg u/\lg^3 n) + d_h \lg u + O(n/\lg^3 n + d_h) \times \lg u + O(n \lg\lg n/\lg^3 n) \times \lg u$ bits, where the inequality follows from Lemma 32.

Next, the extra space needed by the partial rank and select structures is equal to space for rank of the smallest item of each `BSD` + space for $F[1..n/\lg^3 n]$ and k'' + space for $G[1..k''] \leq O(n \lg\lg n/\lg^3 n) \times \lg u +$ $O(n/\lg^3 n) \times \lg u +$ $O(n \lg\lg u/\lg^3 n) \times \lg u$ bits. The total space requirement for all distributors is at most $\sum_{i=1}^{h} (O(n/\lg^3 n) + d_i) \lg u + O(n(\lg\lg n)(\lg u)/\lg^3 n)$ which is $O(n(\lg\lg n)(\lg u)/\lg^3 n) + \sum_{i=1}^{h} d_i \lg u \leq O(n(\lg\lg n)(\lg u)/\lg^3 n)$ bits, where the last inequality is based on Lemma 32.

Finally, since the above space terms can be bounded by $O(n\lg(u/n)/\lg n)$ and the space of all the `BSD` data structures is bounded by *gap* + $O(n \lg\lg(u/n))$ bits (Lemma 31), we have the following theorem.

Theorem 19. *Given a set S of n items from a universe $[1, u]$, we implement an indexable dictionary (ID) in gap$(S) + O(n\lg(u/n)/\lg n) + O(n\lg\lg(u/n))$ bits supporting partial rank and select queries in $O(\lg\lg n)$ time.* □

4.6 Experimental Results

In this section, we present our experimental results, based on the `BSGAP` structure from Corollary 6. Recall that the `BSGAP` structure is organized similarly to a `BSD`, but gap encodes the difference between an item s and its best ancestor $anc(s)$. Section 4.6.1 describes the experimental setup that we use for our results. In Section 4.6.2, we discuss various issues with the space requirements of our `BSGAP` structure and give some intuition about how to encode the various parts of the `BSGAP` structure efficiently. In Section 4.6.3, we describe a further tweakable parameter for our `BSGAP` structure and use it as a black box to succinctly encode blocks of data.

Apart from the δ code, the nibble code [BB04], and the nibble4 code we have mentioned in Section 4.2.1, in this section, we also refer to a number of variations of prefix codes as follows:

- The *delta squared code* encodes the value $\lceil \lg(g_i + 1) \rceil$ using δ codes, followed by the binary representation of g_i. For instance, the delta squared code for 170 is **001 00 1000 10101010**.

- The *nibble4Gamma* encodes the "nibble" part of the nibble4 code using the γ code instead of unary.[8] For instance, the nibble4Gamma code for 170 is **01 0 10101010**.

- In case the universe size of the data set is at most 2^{32}, we will also have the *fixed5 code* which encodes the value $\lceil \lg(g_i + 1) \rceil$ in binary using five bits. For instance, 170 is encoded as **01000 10101010**.

- For larger universe sizes (such as our 2^{64}-sized ones), we use the *nibble4fixed code*, a mix of the nibble4 code and the fixed5 code. Here, we encode the "nibble" part of the nibble4 code using four bits.

For each of these codes, we create a small table of values so that we can decode them quickly when appropriate. As described in Section 4.3, these tables add negligible space, and we have accounted for this (and other) table space in the experimental results that we describe throughout the chapter.

[8]The "nibble" part will be an integer between 1 and 16. The γ code for an integer x is a unary encoding of $\lceil \lg x \rceil$ followed by the binary encoding of x in $\lceil \lg(x + 1) \rceil$ bits.

4.6.1 Experimental Setup

Our source code is written in `C++` in an object-oriented style. The experiments were run on a Dell PowerEdge 650 with 3 GB of RAM. The machine was running Centos 4.1, with a `gnu g++ 3.4.4` compiler. The data sets used were as follows:

- `ip1`: List of IP addresses obtained from Duke University's Computer Science Department. The list refers to 159,690 IP addresses that hit the Duke CS pages in the month of January 2005.
- `ip2`: Similar to `ip1`, but this list consists of 148,700 IP addresses that hit the Duke CS pages in February 2005.
- `upc_32`: List of 100,000 UPC codes obtained from items sold by the Wal-Mart supermarket that fit in a universe of size 2^{32}.
- `isbn`: List of 390,000 ISBNs of books at the Purdue Libraries in a 32-bit format.
- `upc_48`: List of 432,223 UPC codes in the original 48-bit format obtained from items sold by the Wal-Mart supermarket.
- `title`: List of 256,391 book titles from Purdue Libraries, converted into a numeric value out of a universe of size 2^{64}.

4.6.2 Code Comparisons for Encodings and Pointers

We performed experiments to compare the space/time tradeoffs of using different encodings in place of nibble4. We summarize those experiments in Figure 4.3. The figures in the top row show the time required to process 10,000 randomly generated *rank* queries with a `BSGAP` structure using the codes listed, averaged over 10 trials. The figures in the bottom row show the space (in bits) required to encode the `BSGAP` data structure using the listed prefix codes. Each of the bottom two rows also has the information-theoretic minimum and $gap(S)$ listed for reference.

It is clear that both fixed5 and nibble4 are very good codes in the `BSGAP` structure for the 32-bit case; fixed5 is slightly faster than nibble4, and nibble4 is slightly more space-efficient. (For the `isbn` file, nibble4 is significantly more space-efficient.) For 64-bit files, nibble4 is the clear choice. Since our focus is on space efficiency, the rest of the chapter will build `BSGAP` structures with nibble4. (For our 64-bit data sets, we will actually use nibble4fixed.)

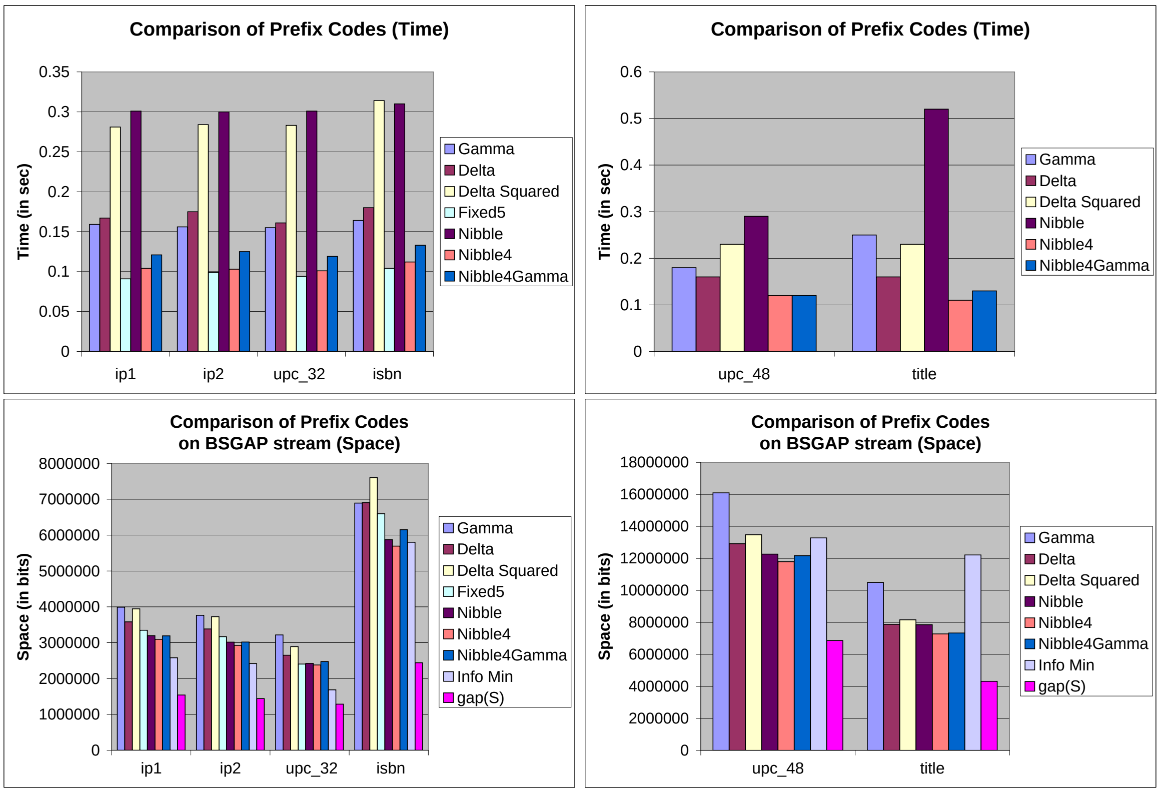


Figure 4.3: Comparison of codes and measures for the data files in Section 4.6.1.

Next, we investigate the cost of these `BSGAP` pointers and see if a different choice of code *for just the pointers* can improve its cost. We summarize the space/time tradeoffs in Figure 4.5. The figure shows the pointer costs (in bits) of each `BSGAP` structure. As we can see, nibble4 and nibble are both space-efficient for the pointer

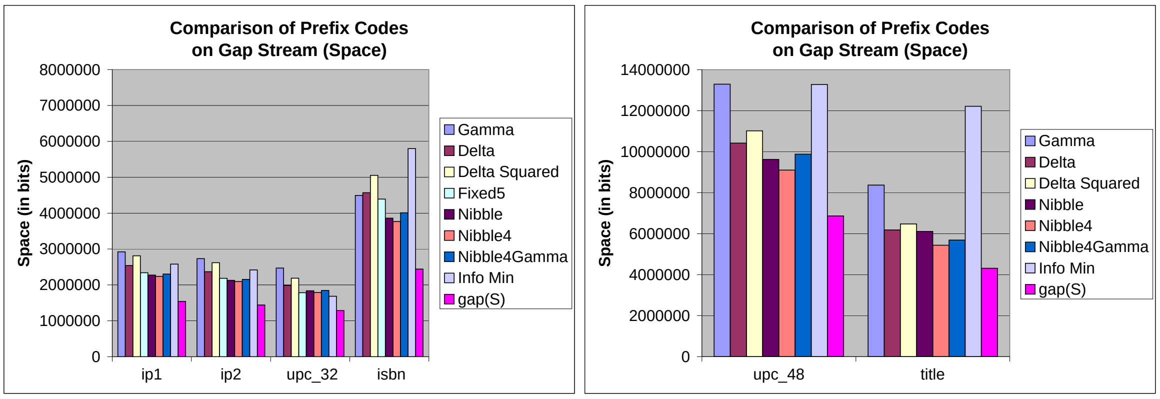


Figure 4.4: Comparison of gap+codes, lg $\binom{u}{n}$, and *gap*(S) for real-data files, described in Section 4.6.1.

distribution. However, nibble4 is again the logical choice, since it is both the most space-efficient and very fast to decode. If we remove these pointer costs from the total space cost for the `BSGAP` structure, we see that this space is *about the same as encoding the gap stream sequentially*; as such, we can think of the pointer overhead for `BSGAP` as a cost to support fast searching.

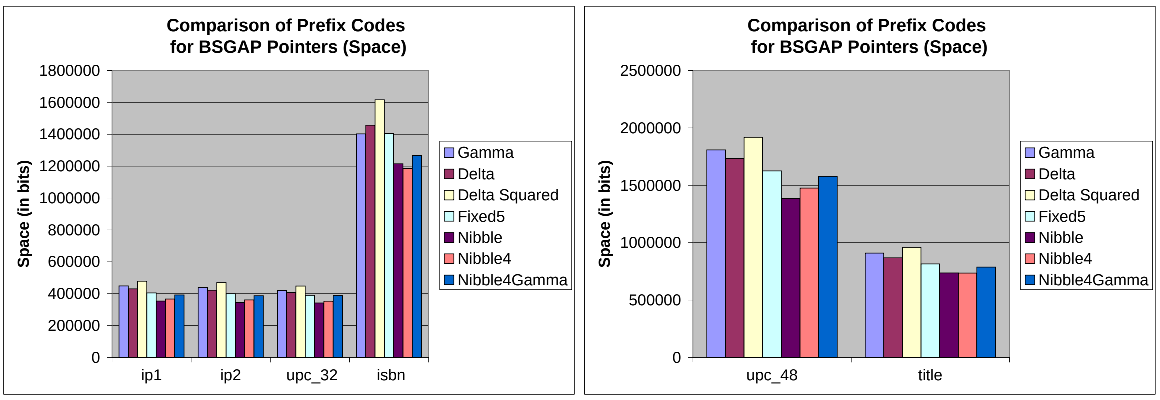


Figure 4.5: Comparison of prefix codes for `BSGAP` pointers for the data files in Section 4.6.1.

4.6.3 `BSGAP`: The Succinct Binary-Searchable Black Box

In this section, we focus on the practical implementation of our fully-indexable dictionary, modeled after Corollary 6. To make our practical dictionary, we replace [AT00] with a simple binary search tree, and introduce a new parameter $h = O(\lg\lg n)$ that does not affect the theoretical time for BSGAP but provides a noticeable improvement in practice. For each group of $\lg^2 n$ items that is stored using BSGAP, we further tune our structure to resort to a simple sequential encoding scheme when there are at most h items left to search, where $h = O(\lg\lg n)$. Theoretically, the time required to search in the BSGAP structure is still $O(\lg\lg n)$. We employ this technique when sequential decoding is fast enough, to avoid writing bits to jump to the right half of the tree. (We call this the *pointer cost.*) In our experiments, we actually let h range up to $\lg^2 n$, to see the point at which a sequential decoding of h items becomes impractical. It turns out that these few adjustments to our theoretical work result in a fast and succinct practical dictionary.

For the rest of the section, we define a parameter b that governs the number of items contained in each BSGAP structure and a parameter h that controls the degree of sequential encoding within a BSGAP data structure, as described above. We denote a particular configuration of our dictionary structure by $D(b, h)$. Let BB refer to the data structure in [BB04]. In this framework, BB is a special case of our dictionary $D(b, h)$ when $h = b$.

In Figure 4.6, we show a space/time tradeoff for BB and our dictionary. Each graph plots space vs. time, where the time is that required to process 10,000 randomly generated *rank* queries, averaged over five trials. Here, we tune BB to operate on the same number of items in each block to avoid extra costs for padding and give them the same benefits as BSGAP receives. For each graph in Figure 4.6, we let the blocksize b range from $[2, 256]$ and the hybrid value range from $[2, b]$. We collect time and space statistics for each $D(b, h)$ data structure. The BB curve is generated from the 256 points corresponding to $D(b, b)$. For the BSGAP curve, we partition the x-axis into 300 partitions and choose the most time-efficient implementation of $D(b, h)$ taking that much space. Notice that our BSGAP structure converges to BB as we allow more space for the data structures, but we have some improvement for extremely small space.

Since BB is a subcase of our `BSGAP` structure, one might think that our space-time curve should never be higher than BB's. However, the curve is generated with actual data structures $D(b, h)$ taking a particular space and time. So, the existence of a point above the BB curve on our `BSGAP` curve simply means that there exists one configuration of our data structure $D(b, h)$ which has those particular results.

The parameter h is crucial to achieving a good space/time tradeoff. Notice that as h increases, the space of $D(b, h)$ decreases because we store fewer pointers in each `BSGAP` data structure. One may think of transferring this saved space into entries in the top level binary search tree to speed up the query time. On the other hand, the time required to search at the bottom of each `BSGAP` structure increases linearly with h. So, there must be some moderate value of h that balances these costs and arrives at the best space/time tradeoff. Hence, we collect all (b, h) pairs and evaluate the best candidates among them.

In Figure 4.7, we compare BB and our dictionary for 64-bit data. We plot space vs. time, where the time is that required to process 1,000 randomly generated *rank* queries, averaged over five trials. We collect data for $D(b, h)$ as before, where the range for b and h for `upc_48` is $[2, 512]$ and `title` is $[2, 2048]$. Notice that our data structure provides a clear advantage over BB as the universe size increases.

4.7 Applications of Succinct Dictionaries

In this section, we describe an application of our FID dictionaries to the case of text indexing. As we mentioned in Sections 3.2.1 and 3.2.2, run-length encoding (RLE) can be a better choice in some applications, particularly when the input set S is dense with respect to its universe U. As a slight deviation from the theme of this chapter,

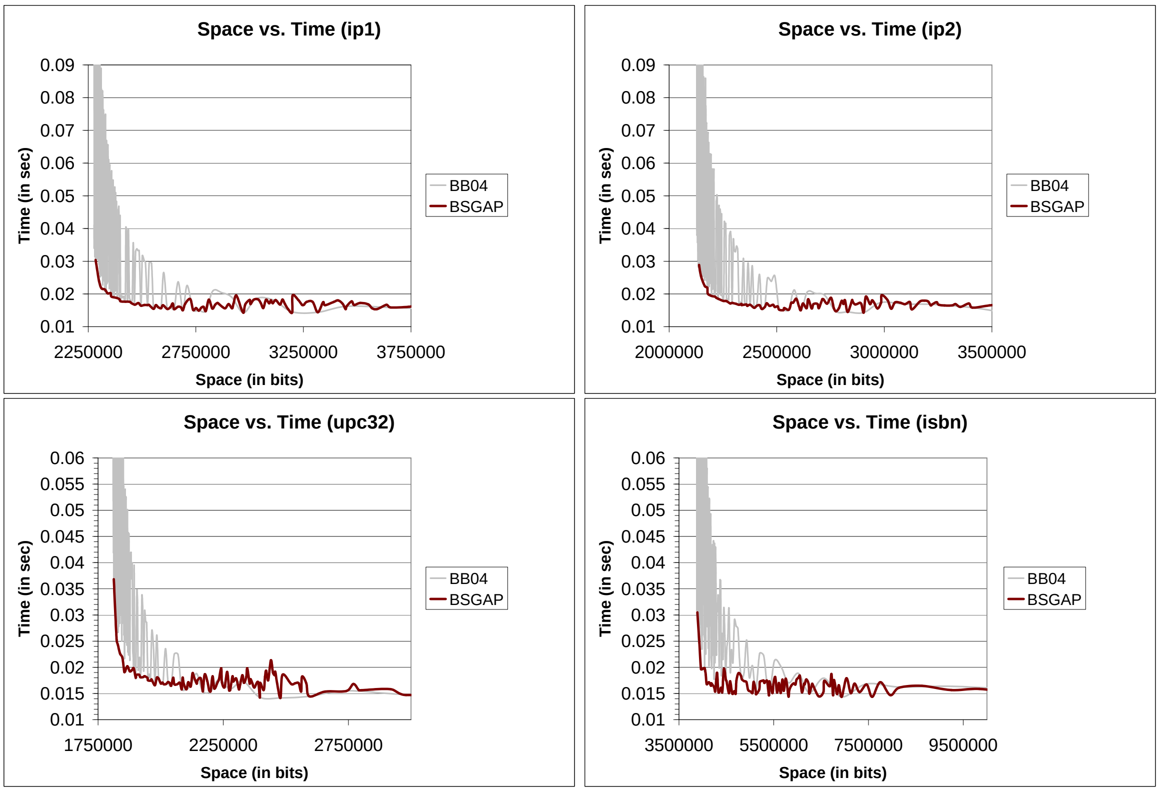


Figure 4.6: Comparison of BB and BSGAP on 32-bit data files in Section 4.6.1.

we consider encoding schemes to manage these dense data sets.

We describe a new practically-motivated data structure called BSRLE that is a modification of our BSGAP structure, but it performs well on dense subsets. We then apply it to text indexes and describe a series of experiments showcasing space/time tradeoffs. In Section 4.7.1, we describe our experimental setup. Section 4.7.2 describes the BSRLE data structure; it improves upon the practical dictionary in Section 3.2.4 in terms of space, based on our discussions in this chapter. Section 4.7.3 presents some results on an improved CSA in comparison with the FM-index [FM05, FM01].

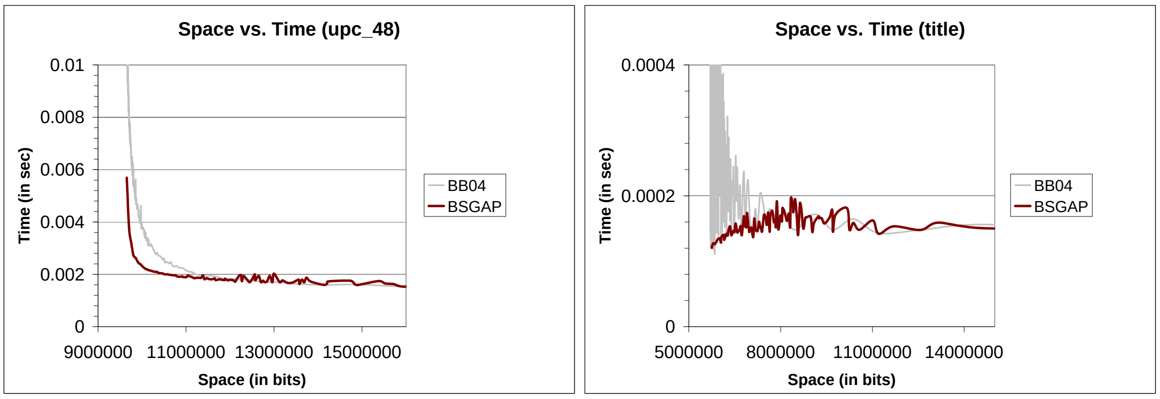


Figure 4.7: Comparison of BB and `BSGAP` on 48-bit and 64-bit data files in Section 4.6.1.

4.7.1 Experimental Setup

Our source code is written in `C++` in an object-oriented style. The experiments were run on a Dell PowerEdge 650 with 3 GB of RAM. The machine was running Centos 4.1, with a `gnu g++ 3.4.4` compiler. We chose data sets that were large enough to observe the space/time tradeoffs, since the minimum indexing overhead can be significant with respect to the file size. (For instance, we use tables to quickly decode our prefix codes, such as nibble4 and the γ code. These tables, which are normally negligible in size for larger files, may be significant for small file sizes.)

- `alice29.txt`: An ASCII version of the book "Alice in Wonderland" from the Canterbury corpus, with an original file size of 152,089 bytes.
- `E.coli`: DNA sequence for the virus E.coli. The original file size is 4,638,690 bytes of space.
- `dblp.50MB`: XML file that provides bibliographic information on major computer science journals and proceedings obtained from `dblp.uni-trier.de`. Downloaded on September 27, 2005 and consisting of exactly 52,428,800 bytes of data.

- `english.50MB`: Concatenation of English text files selected from etext02 to etext05 collections of the Gutenberg Project. The headers from the project were removed, to leave the actual text. Downloaded on May 4, 2005, and consisting of exactly 52,428,800 bytes of data.

4.7.2 Binary Searchable Run-Length Encoding

Before describing our `BSRLE` data structure, we briefly review run-length encoding. We can represent a set S (with n items) out of a universe U of size u using a bitvector B of length u, where each $\mathbf{1}$ represents an item in set S. Run-length encoding represents each subsequence of identical bits (a run) in B as the pair (ℓ, b), where ℓ is the number of times that bit b is repeated. We can avoid encoding b by explicitly storing the first bit, since b will alternate between $\mathbf{0}$ and $\mathbf{1}$. Suppose (without loss of generality) the bitvector B corresponding to the set S is

$$B = \mathbf{0}^{\ell_1}\mathbf{1}^{\ell_2}\mathbf{0}^{\ell_3}\dots\mathbf{1}^{\ell_{2n_\mathbf{1}}},$$

where $n_\mathbf{1}$ is the number of runs of $\mathbf{1}$s in B. We define the *RLE measure* as

$$rle(S) = \sum_{i=1}^{2n_\mathbf{1}} \lceil \lg(\ell_i + 1) \rceil.$$

In the rest of Section 4.7, we will use the γ code to store the length ℓ, since it is useful in the text indexing setting, as shown in Section 3.2.2.

Now we describe how to build the `BSRLE` data structure. We build a modified subset S' of size $n_\mathbf{1}$ corresponding to B. For each run of $\mathbf{1}$s, we add a single *candidate item* to S'. A candidate item r_i is either the first or last $\mathbf{1}$ in run i. (We describe which one to choose when we build `BSRLE`.) We then write the representation `BSRLE`(S), which is a modified version of the encoding of `BSGAP`(S') for the set S'. We reuse the notation for S_L and S_R from `BSD` and `BSGAP`, where S_L contains the subset of items from S from runs 1 to $\lceil n_{\mathbf{1}}/2 \rceil - 1$, and S_R contains the items from runs $\lceil n_{\mathbf{1}}/2 \rceil + 1$ to $n_{\mathbf{1}}$.

The BSRLE encoding is defined as

$$\texttt{BSRLE}(S) = \langle r_{\lceil n_{\mathbf{1}}/2 \rceil} \ominus anc(r_{\lceil n_{\mathbf{1}}/2 \rceil}); \ell_{\lceil n_{\mathbf{1}}/2 \rceil} - 1; p_{\lceil n_{\mathbf{1}}/2 \rceil}; |\texttt{BSRLE}(S_L)|; \texttt{BSRLE}(S_L); \texttt{BSRLE}(S_R)\rangle,$$

where the candidate element $r_{\lceil n_{\mathbf{1}}/2 \rceil}$ is stored using the γ code, $\ell_{\lceil n_{\mathbf{1}}/2 \rceil} - 1$ indicates the number of **1**s in the $\lceil n_{\mathbf{1}}/2 \rceil$th run (not counting the candidate), stored using the γ code, and $p_{\lceil n_{\mathbf{1}}/2 \rceil}$ indicates the number of **1**s in S in the left subtree S_L of the $\lceil n_{\mathbf{1}}/2 \rceil$th run, not counting other candidate items. (In other words, it stores the number of RLE-encoded items that are in S_L.) We store $p_{\lceil n_{\mathbf{1}}/2 \rceil}$ using the nibble4 code.

Now we explain how to choose the candidate element r_i. If $anc(r_i) > r_i$, r_i is the last **1** in run i; otherwise, it is the first **1** in run i. Computing r_i in this way saves space in the encoding, but for ease of exposition, we assume that r_i is the first **1** in run r_i, since the first **1** can easily be determined using the candidate element and ℓ_i. We compute $anc(r_i)$ by building the ancestor set A_i as we did for BSD. However, at each ancestor node r_j (corresponding to the jth run) for r_i, we insert *both* the values corresponding to the first **1** in run r_j and the last **1** in run r_j into the ancestor set A_i.

Given BSRLE(S), $rank(S, a)$ and $select(S, i)$ can be computed in $O(\lg n_{\mathbf{1}})$ time by calling the functions $rrank(\texttt{BSRLE}(S), a, 0, u, n_{\mathbf{1}})$, $rselect_{\mathbf{1}}(\texttt{BSRLE}(S), i, 0, u, n_{\mathbf{1}})$ and $rselect_{\mathbf{0}}(\texttt{BSRLE}(S), i, 0, u, n_{\mathbf{1}})$ detailed below. (As usual, $rank_{\mathbf{0}}(S, a) = a - rank_{\mathbf{1}}(S, a)$.) In the pseudocode, the function *decode_node*(B) returns the values r_i, ℓ_i, and p_i for the ith node. (The techniques used to decode this information are similar to BSD.) The variables *la* and *ra* refer to the left and right ancestors of the current run, respectively.

```
function $rrank(B, a, la, ra, n)$ {
    if $(n = 0)$ return $0$;
    $r, \ell, p \leftarrow decode\_node(B)$;
    if $(a < r)$
     return $rrank(\texttt{BSRLE}(S_L), a, la, r, \lceil n/2 \rceil - 1)$;
    else if $(a < r + \ell)$ return $\lceil n/2 \rceil + p + (a - r)$;
    else return $\lceil n/2 \rceil + p +$
     $rrank(\texttt{BSRLE}(S_R), a, r + \ell - 1, ra, n - \lceil n/2 \rceil)$;
}
function $rselect_1(B, i, la, ra, n)$ {
  $r, \ell, p \leftarrow decode\_node(B)$;
  $c \leftarrow \lceil n/2 \rceil + p$;
  if $(i < c)$
   return $rselect_1(\texttt{BSRLE}(S_L), i, la, r, \lceil n/2 \rceil - 1)$;
  else if $(i < c + \ell)$ return $r + (i - c)$;
  else return
   $rselect_1(\texttt{BSRLE}(S_R), i - c - \ell, r + \ell - 1, ra, n - \lceil n/2 \rceil)$;
}
```

```
function $rselect_{\mathbf{0}}(B, i, la, ra, n)$ {
  $r, \ell, p \leftarrow decode\_node(B)$;
  $c \leftarrow r - (\lceil n/2 \rceil + p)$;
  if $(n = 0)$
    if $(i > c)$
      return $r + (i - c)$;
    if $(i < c)$
      return $la + i$;
  if $(i < c)$
    return $rselect_{\mathbf{0}}(\texttt{BSRLE}(S_L), i, la, r, \lceil n/2 \rceil - 1)$;
  else if $(i > c)$ return
    $rselect_{\mathbf{0}}(\texttt{BSRLE}(S_R), i - c, r + \ell - 1, ra, n - \lceil n/2 \rceil)$;
}
```

Compared to the practical dictionaries from Section 3.2.4, the $\texttt{BSRLE}(S)$ encoding uses the same space to encode the run-length values for **0** and **1**. However, the practical dictionaries store prefix sums for both **0** and **1**, whereas we only store them for **1**s. Moreover, since our prefix sums are localized, we save even more space. In addition, we have a clear space/time tradeoff: our data structure operates in $O(\lg n_{\mathbf{1}})$ time (which is less than `BSGAP`'s $O(\lg n)$ time if $n_{\mathbf{1}}$ is small enough), however, we may spend more time on each step since we decode more γ codes. We summarize its achievements in the following lemma.

Lemma 33 (BSRLE). *The representation* $\texttt{BSRLE}(S)$ *is a fully indexable dictionary (FID) occupying* $rle(S) + O(n \lg\lg(u/n))$ *bits while supporting rank and select functions in* $O(\lg n_{\mathbf{1}})$ *time, where* $n_{\mathbf{1}}$ *is the number of runs of* **1***s in the bitvector representation of* S.

Proof. This proof follows from the proof of Lemma 30 and Theorem 17. Our `BSRLE` encoding achieves $rle(S)$ space by construction, since the run-lengths for the **1**s are stored explicitly, and the run-lengths for the **0**s are stored implicitly by the encoding of BSGAP(S'). The only additional cost we have is to store p_i, which is roughly the number of **1**s in S_L (the left subtree of r_i); the encoding of p_i over all nodes can be bounded by the pointer cost (to jump to the right subtree), and takes at most $O(n)$ bits of space. □

4.7.3 Experimental Results

In this section, we apply our BSRLE data structure to the text indexing problem. In particular, we improve upon the implementation of compressed suffix arrays from Chapter 3 and compare it to the FM-index[FM05, FM01], a state-of-the-art data structure with good theoretical results and practical performance. We make use of the hybrid value h and block length b in tweaking the BSRLE structure, just as we did with BSGAP. Throughout our experiments with BSRLE, we use nibble4 to represent pointers and auxiliary information, and γ codes to represent the actual RLE lengths. For both codes, we maintain a small table of values to facilitate fast decoding; these tables contribute negligible space, and our experimental results account for these costs.

Our goal is to index the text T of length u. We replace each of the practical dictionaries from the earlier CSA implementation (that were used in the wavelet tree) with our new BSRLE dictionaries. This application was the main motivation for developing BSRLE dictionaries. We also redefined the fractional cascading that links these BSRLE dictionaries together to improve the sequential searches in the wavelet tree.

We also drastically speed up the decoding of LCP values that are needed by the CSA. To review, store the LCPs using Sadakane's method [Sad02b]. However, we cannot afford to store all $2u$ bits required. Instead, we store only LCP values larger than $2 \lg u$. To reduce query time, we also store a few dictionaries that keep track of small LCP values. In particular, we maintain a dictionary D_i drawn from a universe of size u, such that its entries correspond to the positions with an LCP value of i. We store a series of these dictionaries $D_1, D_2, \ldots, D_l$, where l is a tweakable parameter that presents space/time tradeoffs.

The LCP lookup proceeds by finding out how many **1** bits appear in D_i within the range corresponding to the two strings in the LCP query. If there are none, we proceed with the search in the next dictionary D_{i+1}. Once we run out of dictionaries to search, we preform an inverse suffix array query (SA^{-1}) to get the location of the two suffixes that start at the lth position in the original query suffixes. Then we reuse our original series of l dictionaries. This process avoids the (relatively) slow lookup time for $\Phi(i)$, at a cost of some additional storage.

We can organize this series of dictionaries D_i in terms of an LCP *wavelet tree*, providing, in theory, many of the benefits we have described earlier for wavelet trees. The main advantage here is in improving the time bound—it's not clear whether an entropy bound makes sense for the storage of LCP values. In practice, short LCP values are much more common and need to be retrieved in $O(1)$ time, rather than the $O(\lg l)$ time for this wavelet tree.

The FM-index uses three parameters in optimization: a two-phase bucketing stage that is similar to our `BSRLE` structure (but lacking the tuned top level with gap encodings), and a frequency percentage f. Suppose f is 2% (the default for the FM-index implementation). The index inserts a special unique symbol at regular intervals in text T such that the total number of symbols is 2% of the text length. This puts a maximum on the number of symbols that the FM-index has to decode, and it addresses the *same* problem that we were trying to address earlier by explicitly storing LCP values. As f increases, the fewer symbol decodings are needed; however, this method requires additional (tuneable) space. We also use this idea when tuning our data structure.

In Figure 4.8, we show a space/time tradeoff for our improved CSA and the FM-index. Each graph plots space vs. time for either *count* or *locate* queries. Each row of graphs shows the results for the files `alice29.txt`, `E.coli`, `dblp.50MB`, and `english.50MB`, respectively. (Each file is described in Section 4.7.1.) We performed *count* and *locate* on $1,000$ randomly generated patterns P, averaged over five trials. The time reported for *count* is the number of milliseconds (msec) required *per symbol* of the input pattern P, and the time for *locate* is the number of milliseconds required per occurrence of P in text T.

To generate each curve in the graphs, we generate all possible data structures using the various parameters for each implementation. Then, we partition the x-axis and chose the most time-efficient implementation of CSA and FM-index taking that much space. Notice that our CSA data structure is competitive with the FM-index for nearly all ranges, although it is slightly slower as we increase the space allowed. However, what is most interesting is its behavior when we allow a minimum of extra bits of space. For this case, our data structure presents the fastest implementation for extremely succinct space.

4.8 Conclusions

In this chapter, we have formalized and developed measures for analyzing the space needed to store set data. These measures can provide a framework for further investigation of compressed data structuring techniques. We have achieved a fully indexable dictionary that operates in near-optimal time ($AT(u,n)$) to support rank, select, and predecessor queries, while just taking $\mathit{gap} + O(n \lg(u/n)/\lg n) + O(n \lg\lg(u/n))$ bits of storage. This result improves a number of compressed data structures [RRR02, AT00, BB04] by reducing space usage, while maintaining nearly-optimal time bounds. Our *gap* term has a constant of 1, which is extremely important when considering matters of space efficiency. Equally important are the properties of the other space terms—if $n = o(u)$, they amount to $o\left(\lg \binom{u}{n}\right)$ bits. Also, our dictionary is the first that achieves $O(n \lg(u/n))$ bits of space, without significantly sacrificing the query times. (Recall that we take $AT(u, n) \geq BF(u, n)$ time.) We also provide an indexable dictionary which operates in $gap + O(n \lg(u/n)/ \lg n) + O(n \lg \lg(u/n))$ bits and supports queries in $O(\lg \lg n)$ time. We conjecture that if the space for an ID is measured in terms of *gap*, $O(1)$ query time may not be possible to achieve. Since the *gap* measure inherently exploits the encoding of items with respect to other items, $O(1)$ decoding time of an item (and thus searching) is not straightforward.

In addition, we have shown evidence that data-aware measures (such as *gap*) tend to be smaller than combinatorial measures on real-life data. Employing techniques that exploit the redundancy of the data can lead to more succinct data structures and a better understanding of the underlying information. As such, we encourage researchers to develop theoretical results with a data-aware analysis. In particular, our `BSGAP` data structure, along with BB (proposed in [BB04]) are extremely succinct in practice for sparse data sets. In addition, we provide some evidence that `BSGAP` is less sensitive than [BB04] to an increase in the size of the universe. Finally, we provide some useful information on the relative performance of prefix codes with respect to compression space and decompression time.

There are two open problems. Is it possible to give an indexable dictionary with query times further reduced, and with space measured in a data-aware manner? Another problem is whether we can extend our data structures to support dynamic operations.

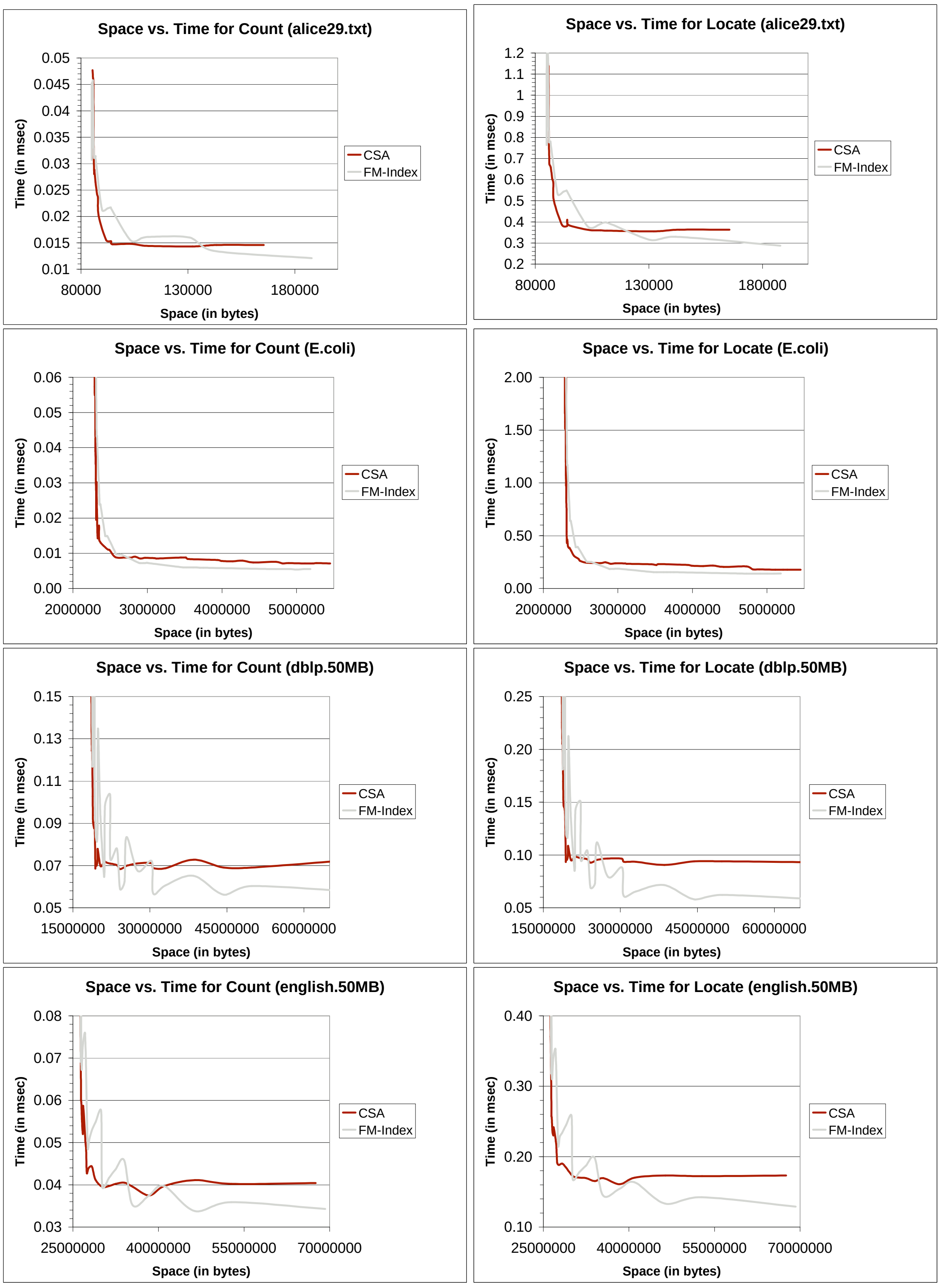


Figure 4.8: Comparison of CSA and FM-index on count and locate.

Chapter 5

Dynamizing Succinct Data Structures

5.1 Introduction

The new trend in indexing data structures is to compress and index data in one shot. The ultimate goal of these compressed indexes is to retain near-optimal query times (as if not compressed), yet still take near-optimal space (as if not an index). A few pioneer results are [GV00, GGV03, FM05, RRR02, GMR06, FLMM05]; there are many others. For compressed text indexing, see Navarro and Mäkinen's excellent survey [NM06a].

Progress in compressed indexing has also expanded to more combinatorial structures, such as trees and subsets. For these *succinct* data structures, the emphasis is to store them in terms of the information-theoretic (combinatorial) minimum required space with fast query times [RRR02, Jac89b, HMP01]. Compressed text indexing makes heavy use of succinct data structures for set data, or *dictionaries.*

The vast majority of succinct data structuring work is concerned largely with static data. Although the space savings is large, the main deterrent to a more ubiquitous use of succinct data structures is their notable lack of support for dynamic operations. Many settings require indexing and query functionality on dynamic data: XML documents, web pages, CVS projects, electronic document archives, etc. For this type of data, it can be prohibitively expensive to rebuild a static index from scratch each time an update occurs. The goal is then to answer queries efficiently, perform updates in a reasonable amount of time, and *still* maintain a compressed version of the dynamically-changing data.

In that vein, there have been some results on dynamic succinct bitvectors (dictionaries) [RRR01, HSS03, NM06b]. However, these data structures either perform queries in far from optimal time (in query-intensive environments), or allow only a limited range of dynamic operations ("flip" operations only). Here, we consider the more general update operations consisting of arbitrary insertion and deletion of bits, which is a central challenge in dynamizing succinct data structures for a variety of applications. We define the *dynamic text dictionary* problem: Given a dynamic text T of n symbols drawn from an alphabet Σ, construct a data structure (index) that allows the following operations for any symbol $s \in \Sigma$:

- $\mathit{rank}_s(i)$ tells the number of s symbols up to the ith position in T;
- $\mathit{select}_s(i)$ gives the position in T of the ith s;
- $\mathit{char}(i)$ returns the symbol in the ith position of T;
- $\mathit{insert}_s(i)$ inserts s before the position i in T;
- $\mathit{delete}(i)$ deletes the ith symbol from T.

When $|\Sigma| = 2$, the above problem is called the *dynamic bit dictionary* problem. For the static case, [RRR02] solves the bit dictionary problem using $nH_0 + o(n)$ bits of space and answers rank and select queries in $O(1)$ time, where H_0 is the 0th order empirical entropy of the text T. The best known time bounds for the dynamic problem are given by [NM06b], achieving $O(\lg n)$ for all operations.[1]

The text dictionary problem is a key tool in text indexing data structures. For the static case, Grossi et al. [GGV03] present a wavelet tree structure that answers queries in $O(\lg|\Sigma|)$ time and takes $nH_0 + o(n \lg|\Sigma|)$ bits of space. Golynski et al. [GMR06] improve the query bounds to $O(\lg\lg|\Sigma|)$ time, although they take more space, namely, $n\lg|\Sigma| + o(n\lg|\Sigma|)$ bits of space. Nevertheless, their data structure presents the best query bounds for this problem.

Developing a dynamic text dictionary based on the wavelet structure can be done readily using dynamic bit dictionaries (as is done in [NM06b]) since updates to a particular symbol s only affect the data structures for $O(\lg|\Sigma|)$ groups of symbols according to the hierarchical decomposition of the alphabet Σ. The solution to this problem is given by Mäkinen and Navarro [NM06b], with an update/query bound of $O(\lg n \lg|\Sigma|)$. These bounds are far from optimal, especially in query-intensive settings. On the other hand, the best known query

[1]There is another data structure proposed in [HSS03], requiring non-succinct space.

bounds for static text dictionaries are given by [GMR06], which treats each symbol in Σ individually; an update to symbol s could potentially affect Σ different data structures, and thus may be hard to dynamize.

We list the following contributions of this chapter:

- We develop a general framework to dynamize many succinct data structures like ordinal trees, labeled trees, dictionaries, and text collections. Our framework can transform any static succinct data structure D for a text T into a dynamic succinct data structure. Precisely, if D supports $rank_s$, $select_s$, and *char* queries in $O(t(n))$ time and takes $s(n)$ bits of space, the dynamic data structure supports queries in $O(t(n) + \lg\lg n)$ time and updates in amortized $O(n^\epsilon)$ time and takes just $s(n) + o(n)$ bits of space.
- Our results represent near-optimal tradeoffs for update/query times for the dynamic text (and bit) dictionary problem. (For lower bound, see [PD06].)
- We provide the first succinct data structure for the *dynamic bit dictionary* problem. Our data structure takes $nH_0 + o(n)$ bits of space and requires $O(\lg\lg n)$ time to support $rank_s$, $select_s$, and *char* queries while supporting updates to the text T in amortized $O(n^\epsilon)$ time.
- We provide the first near-optimal result for the *dynamic text dictionary* problem on a dynamic text T. Our data structure requires $n \lg |\Sigma| + o(n \lg |\Sigma|)$ bits of space and supports queries in $O(\lg\lg n)$ time and updates in $O(n^\epsilon)$ time. When $|\Sigma| = \text{polylg}(n)$, we can improve our query time to $O(1)$.
- Our framework can dynamize succinct data structures for labeled trees, text collections, and XML documents.

The work done in this chapter is a collaborative effort with Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter.

5.1.1 Outline

In Section 5.2, we summarize some existing results including the RRR data structure [RRR02], some static text dictionaries [GGV03, GMR06], and some brief construction bounds. Section 5.3.1 describes our *BitIndel* data structure, which solves the dynamic bit dictionary problem. Section 5.3.3 describes the first part of our dynamic text dictionary; we describe *inX*, which keeps track of where the original text T has been updated. In Section 5.3.4, we then describe *onlyX*, which actually stores the updates themselves. The onlyX structure is a non-succinct data structure of independent interest that solves the dynamic text dictionary problem. In Section 5.5, we apply our dynamic bit and text dictionaries to dynamize ordinal trees, labeled trees, and the XBW transform [FLMM05].

5.2 Preliminaries

We summarize several important static structures that we will use in achieving the dynamic results. The proofs of their construction are omitted due to space constraints. In the rest of this chapter, we refer to a static bit or text dictionary D, that requires $s(n)$ bits and answers queries in $t(n)$ time.

Lemma 34 ([RRR02]). *For a bitvector (i.e., $|\Sigma| = 2$) of length n, there exists a static data structure D called* RRR *solving the* bit dictionary *problem supporting rank, select, and char queries in $t(n) = O(1)$ time using $s(n) = nH_0 + O(n \lg\lg n / \lg n)$ bits of space, while taking only $O(n)$ time to construct.* □

Lemma 35 (Section 2.4.3). *For a text T of length n drawn from alphabet Σ, there exists a static data structure D called the* wavelet tree *solving the* text dictionary *problem supporting rank$_s$, select$_s$, and char queries in $t(n) = O(\lg |\Sigma|)$ time using $s(n) = nH_0 + o(n \lg |\Sigma|)$ bits of space, while taking $O(nH_0)$ time to construct. When $|\Sigma| = \text{polylg}(n)$, we can support queries in $t(n) = O(1)$ time.* □

Lemma 36 ([GMR06]). *For a text T of length n drawn from alphabet Σ, there exists a static data structure D called* GMR *that solves the* text dictionary *problem supporting* *select$_s$ queries in $t_1(n) = O(1)$ time and rank and char queries in $t_2(n) = O(\lg\lg|\Sigma|)$ time using $s(n) = n\lg|\Sigma| + o(n\lg|\Sigma|)$ bits of space, while taking $O(n\lg n)$ time to construct.* $\square$

We also use the following static data structure called prefix-sum (PS) as a building block for achieving our dynamic result. Suppose we are given a non-negative integer array $A[1..t]$ such that $\sum_i A[i] \leq n$. We define the partial sums $P[i] = \sum_{j=1}^{i} A[i]$. Note that P is a sorted array, such that $0 \leq P[i] \leq P[j] \leq n$ for all $i < j$. A prefix-sum (PS) structure on A is a data structure that supports the following operations:

- *sum*(j) returns the partial sum $P[j]$;
- *findsum*(i) returns the index j such that *sum*$(j) \leq i <$ *sum*$(j+1)$.

To support *sum*, we simply store array P explicitly, requiring $O(t\lg n)$ bits of space. To support *findsum*, we take the t prefix sums and cluster them into consecutive groups of size $O(\lg^2 n)$. Within a group, we use a balanced binary search tree to support *findsum* in $O(\lg\lg n)$ time in the standard way. Now we must determine which group to search for a given query. From each of the $O(t/\lg^2 n)$ groups, we store the largest prefix sum using a hashing implementation of a van Emde Boas (VEB) data structure. For the hashing, we use [HMP01] (Theorem 1.1), so that we can construct the hash table deterministically in $O(t)$ time and taking $O(t)$ bits of space. Along with each entry in the hash table, we also store a pointer to its associated group to search further. To answer *findsum*(i), we search the VEB structure to find the right group in $O(\lg\lg n)$ time. We then follow the pointer to the binary search tree and spend an additional $O(\lg\lg n)$ time.

Using [HSS03], we can support *findsum*(i) in $O(1)$ time in the special case where each array entry $A[j]$ is between x and cx; c is a positive constant integer and x is a positive integer. We briefly sketch the idea now. To support *findsum*, we partition the universe n into n/x blocks of length x. Since each $A[j]$ ranges from x to cx, the partial sums $P[j]$ are within c blocks of one another. Thus, $n/x = ct$. For the jth block, we explicitly store $B[j] =$ *findsum*(xj) using $O(\lg t)$ bits. To answer *findsum*(i), we first navigate to the $\lceil i/x \rceil$th block and retrieve the explicit solution $r = B[\lceil i/x \rceil]$ contained there. If $P[r+1] \leq i$, we return $r+1$. Otherwise, we know that we are within x of the correct prefix sum and we return r (because $P[r+1] - P[r] \geq x$). We will require $O(ct \lg t)$ bits of space to store the array B. Thus, we can write the following lemma.

Lemma 37. *Let $A[1 \ldots t]$ be a non-negative integer array such that $\sum_i A[i] \leq n$. There exists a data structure PS on A that supports sum and findsum in $O(\lg \lg n)$ time using $O(t \lg n)$ bits of space and can be constructed in $O(t)$ time. In the particular case where $x \leq A[i] \leq cx$ for all i, where x is a positive integer and $c \geq 1$ is a positive constant integer, sum and findsum can be answered in $O(1)$ time.* □

Proof. The proof follows from the above discussion, where we explicitly store the array P and the array B for each of the ct blocks. □

We also make use of a data structure called the Weight Balanced B-tree (WBB tree), which was used in [RRR01, HSS03]. We use this structure with Lemma 37 to achieve $O(1)$ time. A WBB tree is a B-tree defined with a *weight-balance condition.* A weight-balance condition means that for any node v at level i, the number of leaves in v's subtree is between $0.5b^i + 1$ and $2b^i - 1$, where b is the fanout factor. Insertions and deletions on the WBB tree can be performed in amortized $O(\lg_b n)$ time while maintaining the weight-balance condition.

We use the WBB tree since it ensures that $x \leq A[i] \leq cx$ where c is a positive constant integer, thus allowing constant-time search at each node. However, a simple B-tree would require $O(\lg \lg n)$ time in this situation. Also, WBB trees are a crucial component of the onlyX structure, described in Section 5.3.4. WBB trees are also used in Section 5.3.1 (although B-trees could be used here).

We define a weight balanced B-tree as follows: all leaves of the WBB tree are considered to be at level 0. A level-i node is connected to its parent node at level $i+1$. We define a *weight-balance condition*, such that for any node v at level i, the number of leaves in v's subtree is between $0.5b^i + 1$ and $2b^i - 1$, where b is the fanout factor. Thus, the degree of an internal node is $\Theta(b)$ (from b to $4b$), such that the height of the tree is $\Theta(\lg_b n')$, where n' is the number of leaves in the current tree.

After a leaf is inserted into the tree, the weight-balance condition of some level-i ancestor of the leaf, say v, may be violated. Precisely, this case happens when the number of leaves in v's subtree is $2b^i$. In this case, v will be split into two new nodes at the same level (called a *split* operation), each of them becoming the root of a perfect subtree with b^i leaves. (This split could cause a restructuring of the entire subtree that was split, but this follows standard techniques.)

On the other hand, in case a leaf is deleted, the weight-balance condition of v at level i may be violated; that is, the number of leaves in v's subtree becomes $0.5b^i$. In this case, v is merged with one of its neighboring siblings, and there will be two cases:

(i) if the total number of leaves after merging is less than $1.5b^i$, the update finishes (called a *merge* operation);

(ii) otherwise, the merged node is further split into two nodes, each of them becoming the root of a subtree with half the number of leaves (called a *merge-then-split* operation).

Based on the above updating process, we have the following lemma and corollary.

Lemma 38. *Except the root, when a node v at level i violates the weight-balance condition, at least $\Theta(b^i)$ leaves are inserted or deleted in v's subtree since the creation of v.*

Proof. A node is created when there is either a split, merge, or merge-then-split event. As a result, node v contains at least $0.75b^i$ leaves (by merge-then-split) and at most $1.5b^i$ leaves at its creation. Thus, at least $0.25b^i$ leaves are deleted or at least $0.5b^i$ leaves are inserted before v can violate the weight-balance condition. □

Corollary 9. *Suppose that c_i is the maximum cost of a split, a merge, or a merge-then-split operation when a level-i node violates the weight-balance condition. The amortized cost for supporting the above operations due to an insertion or deletion of a leaf is at most $\Theta(\sum_{i=1}^{h} c_i/b^i)$, where h denotes the current height of the tree.*

Proof. We prove this result by a simple accounting method. A node is created with zero tokens; when a leaf is inserted or deleted, it gives each of its level-i ancestors $\Theta(c_i/b^i)$

tokens (precisely, $4c_i/b^i$ tokens for deletion and $2c_i/b^i$ tokens for insertion). Thus, the total number of tokens given is $\Theta(\sum_{i=1}^{h} c_i/b^i)$ during an insertion or deletion operation. It is easy to verify that there are at least c_i tokens when a node at level i violates the weight-balance condition. In other words, an amortized cost of $\Theta(\sum_{i=1}^{h} c_i/b^i)$ for leaf insertion or deletion is enough to support split, merge, or merge-then-split operations. $\square$

5.3 Data Structures

There are several data structures that support $rank_s$ and $select_s$ queries. They are broadly based on two different approaches: *logarithmic*, which creates a binary search tree with a height of $\lg|\Sigma|$ with each symbol's occurences stored in the leaves; and *log-logarithmic*, which is based on predecessor search and VEB. Despite the faster access of the log-logarithmic approach, it is difficult to update since each symbol $s \in \Sigma$ is treated separately and updating one symbol will affect the data structure for all other symbols. In contrast logarithmic approaches need only manage updates in a particular root-to-leaf path of their binary search tree, so that only $O(\lg|\Sigma|)$ internal nodes are affected for each update.

Our solution is built with three main data structures:

- *BitIndel*: bitvector supporting insertion and deletion, described in Section 5.3.1;
- *StaticRankSelect*: static text dictionary structure supporting $rank_s$, $select_s$, and *char* on a text T;
- *onlyX*: non-succinct dynamic text dictionary, described in Section 5.3.4.

We use StaticRankSelect to maintain the original text T; we can use any existing structure such as GGV or GMR mentioned in Section 5.2. For ease of exposition, unless otherwise stated, we shall use GMR [GMR06] in this section. We keep track of newly inserted symbols N in onlyX such that after every $O(n^{1-\epsilon}\lg n)$ update operations performed, updates are merged with the StaticRankSelect structure. Thus, onlyX never contains more than $O(n^{1-\epsilon}\lg n)$ symbols. We maintain onlyX using $O(n^{1-\epsilon}\lg^2 n) = o(n)$ bits of space. Finally, since merging N with T requires $O(n\lg n)$ time, we arrive at an amortized $O(n^{\epsilon})$

time for updating these data structures. BitIndel is used to translate positions p_t from the old text T to the new positions $p_{\hat{t}}$ from the current text $\hat{T}$. (We maintain $\hat{T}$ implicitly through the use of BitIndel structures, StaticRankSelect, and onlyX.)

5.3.1 Bitvector Dictionary with Indels: BitIndel

In this section, we describe a data structure (BitIndel) for a bitvector B of original length n that can handle insertions and deletions of bits anywhere in B while still supporting *rank* and *select* on the updated bitvector B' of length n'. The space of the data structure is $n'H_0 + o(n')$. When $n' = O(n)$, our structure supports these updates in $O(n^{\epsilon})$ time and *rank* and *select* queries in $O(\lg \lg n)$ time. (In [HSS03], Hon et al. propose a non-succinct BitIndel structure taking $n' + o(n')$ bits of space.)

Formally, we define the following update operations that we support on the current bitvector B' of length n':

- $insert_b(i)$ inserts the bit b in the ith position;
- $delete(i)$ deletes the bit located in the ith position;
- $flip(i)$ flips the bit in the ith position.

For bitvector B', we construct a B-tree $\mathcal{T}$ with fanout between $[n^{\epsilon}, 2n^{\epsilon}]$, for a fixed $\epsilon > 0$. The leaves of $\mathcal{T}$ maintain contiguous chunks of B' ranging from $[n^{\epsilon}, 2n^{\epsilon}]$ in size, such that the ℓth (leftmost) leaf corresponds to the ℓth chunk of B'. Each leaf ℓ maintains an RRR [RRR02] data structure $\ell.R$ that answers *rank* and *select* queries on its $O(n^{\epsilon})$-sized chunks in $O(1)$ time. Each internal node v of $\mathcal{T}$ maintains three arrays: $count_{\mathbf{0}}$, $count_{\mathbf{1}}$, and *size*. Let c_j denote the jth child node of v. The entry $count_{\mathbf{0}}[j]$ is the number of **0**s in the part of the bitvector in the subtree of c_j. The entry $count_{\mathbf{1}}[j]$ is the number of **1**s in the part of the bitvector in the subtree of c_j. The entry $size[j]$ is the total number of bits in the subtree of c_j. To have fast access to this information at each node, we build a PS structure on this information. (We don't actually store $count_{\mathbf{0}}$, $count_{\mathbf{1}}$, and *size* explicitly; rather, we store a PS structure for each array.)

The height of this tree is $O(\lg_n n')$. To traverse down to a leaf for any operation, we use the PS structure at a node (using $O(\lg\lg n)$ time) to determine the next node to visit on the root-to-leaf path. Then, we query our RRR [RRR02] data structure $\ell.R$ at leaf ℓ and return the answer. Now we describe our operations in more detail.

```
function $v.rank_s(i)$ {
  if ($leaf(v)$) return $v.R.rank_s(i)$;
  $j \leftarrow v.size.findsum(i)$;
  return $v.count_s.sum(j)+$
    $c_{j+1}.rank_s(i - v.size.sum(j))$;
}
```

```
function $v.select_s(i)$ {
  if ($leaf(v)$) return $v.R.select_s(i)$;
  $j \leftarrow v.count_s.findsum(i)$;
  return $v.size.sum(j)+$
    $c_{j+1}.select_s(i - v.count_s.sum(j))$;
}
```

Let r be the root node of $\mathcal{T}$. Then, $rank_s(i)$ is answered by invoking $r.rank_s(i)$, and $select_s(j)$ is answered by invoking $r.select_s(j)$.

Time Bounds. Each of the $rank_s$ and $select_s$ queries requires $O(\lg\lg n)$ time per node traversed in the B-tree $\mathcal{T}$. Since there are at most $O(\lg_n n')$ such nodes before encountering a leaf, the total time is $O((\lg_n n')\lg\lg n)$.

Updates. The $flip(i)$ operation can be supported by performing a constant number of *insert*, *delete*, and *rank* operations. So, for updates, we consider only *insert* and *delete*. At every update operation, we traverse the B-tree as before. The prefix-sum data structures in each internal node along the path are rebuilt in $O(n^\epsilon)$ time per node. At the leaf, R is rebuilt. If the leaf node manages more than $2n^\epsilon$ symbols or less than n^ϵ, we invoke the standard B-tree merge/split routines, propagating them up the tree as appropriate. In the worst case, updates take $O(n^\epsilon \lg_n n')$.

Space. There are at most $O(n'/n^{2\epsilon})$ internal nodes (recall that each leaf in the tree corresponds to a chunk of $O(n^\epsilon)$ bits), each taking $O(n^\epsilon \lg n')$ bits. Thus, the total space for the internal nodes is $O((n'/n^\epsilon)\lg n')$. Let $n_\mathbf{1}$ be the number of **1**s in B'. The space for the bottom-level R structures can be bounded by $\lceil \lg \binom{n'}{n_\mathbf{1}} \rceil + o(n')$ bits. As seen in [GGV03], we can write the contribution as $n'H_0 + o(n')$ bits.

Lemma 39. *Given a bitvector B' with length n' and original length n, we can create a data structure that takes $n'H_0 + o(n')$ bits and supports rank and select in $O((\lg_n n') \lg\lg n)$ time, and indel in $O(n^\epsilon \lg_n n')$ time. When $n' = O(n)$, our time bounds become $O(\lg\lg n)$ and $O(n^\epsilon)$ respectively.* □

The prefix sum data structure used inside the B-tree is the main bottleneck to query times, allowing us only $O(\lg\lg n)$ time access. However, if we store three WBB-trees, then separately in each of them the special condition from Lemma 37 can be met allowing us $O(1)$ queries on prefix sum structures. We describe this result in the following section.

5.3.2 Constant-Time BitIndel

In this section, we describe a constant-time query BitIndel data structure for bitvector B of original length n that can handle insertions and deletions of bits anywhere in B while still supporting *rank* and *select* on the updated bitvector B' of length n'. When $n' = O(n)$, our structure supports these updates in $O(n^\epsilon)$ time and *rank* and *select* queries in $O(1)$ time.

We modify BitIndel to perform $O(\lg_n n')$ query time by taking three times as much space, i.e., $3nH_0 + o(n)$ bits. We briefly overview the scheme and the results and then give the details. Instead of a single B-tree, we store three WBB trees, weight balanced by *size*, *count*$_0$, and *count*$_1$. With this new design, both *sum* and *findsum* queries within a node can be performed in $O(1)$ time as each array entry $A[i]$ of the corresponding *size*, *count*$_0$, and *count*$_1$ arrays is between x and $2x$ for some non-negative integer x [HSS03]. The *rank* queries will be answered using the WBB for *size*, while *select*$_s$ will be answered with the WBB for *count*$_s$.

For bitvector B', we construct three WBB Trees U, V, W whose leaves maintain contiguous chunks of B', such that the ℓth (leftmost) leaf corresponds to the ℓth chunk of B'. For the moment, assume that each leaf ℓ maintains its associated chunk of B' explicitly. The internal leaves of U, V, W each maintain the three arrays, *count*$_{\mathbf{0}}$, *count*$_{\mathbf{1}}$, and *size*. (Definitions are similar to above.) However, U is weight-balanced on *count*$_{\mathbf{0}}$, V is weight-balanced on *count*$_{\mathbf{1}}$, and W is weight-balanced on *size*. To summarize, we have the following trees:

- WBB tree U, where the internal node v is weight-balanced on the array $\mathit{count}_{\mathbf{0}}$;
- WBB tree V, where the internal node v is weight-balanced on the array $\mathit{count}_{\mathbf{1}}$; and
- WBB tree W, where the internal node v is weight-balanced on the array *size*.

Queries. Queries are performed as usual, where W answers rank_s queries by traversing according to the array *size* and returning rank_s information by performing *sum* on the count_s array in internal nodes, plus the *rank* information from the explicitly-stored chunk of B' at the leaf. For select_s, we consult the WBB tree storing count_s and return select_s information by performing *sum* on the *size* array in internal nodes, plus the *select* information from the explicitly-stored chunk of B' at the leaf. The queries at each level can be done in constant time using Lemma 37.

Updates. Here, we have to update all three trees. Without loss of generality, suppose we delete a **1**.

- Traverse W by *size* to the appropriate leaf node and compute $\mathit{rank}_{\mathbf{0}}$ and $\mathit{rank}_{\mathbf{1}}$. Then traverse upwards, decrementing the values of $\mathit{count}_{\mathbf{1}}$ and *size* appropriately.
- Traverse U by $\mathit{count}_{\mathbf{0}}$ using the $\mathit{rank}_{\mathbf{0}}$ computed in the previous step to arrive at a leaf. Then traverse upwards, decrementing the values of *size* and $\mathit{count}_{\mathbf{1}}$ appropriately.
- Traverse V by $\mathit{count}_{\mathbf{1}}$ using the $\mathit{rank}_{\mathbf{1}}$ computed in the previous step to arrive at a leaf. Then traverse upwards, decrementing the values of *size* and $\mathit{count}_{\mathbf{1}}$ appropriately.

Apart from these updates at non-leaf levels, we need to reconstruct the RRR data structures stored at leaf-level of the WBB tree also. This can be done easily for W in $O(n^\epsilon)$ time. However, for the structures U (and V) which is weight balanced by count_0 (resp. count_1) the leaf level bitvector stored using RRR can be a lot longer than n^ϵ bits although it is guaranteed to have only $O(n^\epsilon)$ **0**s. In such a case, reconstructing RRR structure can take conceivably a lot more time. We propose a following fix for this situation. Whenever the length of the bitvector stored is more than $O(n^\epsilon \lg^2 n)$ bits we explicitly write the positions of **0**s in an array rather than storing RRR structures. Since the structure U (and V) is select only, the query can be easily answered by constant time array lookup. Since bit vector of length greater than $n^\epsilon \lg^2 n$ is encoded using $n^\epsilon \lg n$ bits, the total space for such explicit encodings throughout the structure can be captured by $o(n')$ term. Now updates of RRR structures can be done in $O(n^\epsilon \text{polylg}(n))$. This can be adjusted by using slightly smaller ϵ.

Space. Since, we store three structures here (instead of one) the space is $3n'H_0 + o(n')$ bits. The rest of the analysis is exactly the same as in the previous subsection and also the space for explicit array encodings (instead of RRR) can be captures by $o(n')$ term.

Lemma 40. *Given a bitvector B' with length n' and original length n, we can create a data structure that takes $3n'H_0 + o(n')$ bits and supports rank and select in $O(\lg_n n')$ time, and indel in $O(n^\epsilon \lg_n n')$ amortized time. When $n' = O(n)$, our time bounds become $O(1)$ and $O(n^\epsilon)$ respectively.* □

If we change our BitIndel structure such that the bottom-level RRR [RRR02] data structures are built on $[\lg^2 n, 2\lg^2 n]$ bits each and set the B-tree fanout factor $b = 2$, we can obtain $O(\lg n)$ update time with $O(\lg n)$ query time. In this sense, our BitIndel data structure is a generalization of [NM06b].

5.3.3 Insert-X-Delete-any: inX

Let $\mathtt{x}$ be a symbol other than those in alphabet Σ. In this section, we describe a data structure on a text T of length n supporting $rank_s$ and $select_s$ that can handle $delete(i)$ and $insert_{\mathtt{x}}(i)$. That is, only x can be inserted to T, while any characters can be deleted from T. Notice that insertions and deletions will affect the answers returned for symbols in the alphabet Σ. For example, T may be abcaab, where $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Here, $rank_{\mathtt{a}}(4) = 2$ and $select_{\mathtt{a}}(3) = 5$. Let $\hat{T}$ be the current text after some number of insertions and deletions of symbol x. Initially, $\hat{T} = T$. After some insertions, the current $\hat{T}$ may be axxxbcaxabx. Notice that $rank_{\mathtt{a}}(4) = 1$ and $select_{\mathtt{a}}(3) = 9$. We represent $\hat{T}$ by the text T', such that when the symbols of the original text T are deleted, each deleted symbol is replaced by a special symbol d (whereas if x is deleted, it is just deleted from T'). Continuing the example, after some deletions of symbols from T, T' may be `axxxddaxabx`. Notice that $rank_{\texttt{a}}(4) = 1$ and $select_{\texttt{a}}(3) = 7$.

We define an *insert vector* I such that $I[i] = \mathbf{1}$ if and only if $T'[i] = \texttt{x}$. Similarly, we define a *delete vector* D such that $D[i] = \mathbf{1}$ if and only if $T'[i] = \texttt{d}$. We also define a delete vector D_s for each symbol s such that $D_s[i] = \mathbf{1}$ if and only if the ith s in the original text T was deleted. The text T' is merely a conceptual text: we refer to it for ease of exposition but we actually maintain $\hat{T}$ instead.

To store $\hat{T}$, we store T using the StaticRankSelect data structure and store all of the I, D, D_s bitvectors using the constant time BitIndel structure. Now, we describe $\hat{T}.insert_{\texttt{x}}(i)$, $\hat{T}.delete(i)$, $\hat{T}.rank_s(i)$, and $\hat{T}.select_s(i)$:

$\hat{T}.insert_{\texttt{x}}(i)$. First, we convert position i in $\hat{T}$ to its corresponding position i' in T' by computing $i' = D.select_{\mathbf{0}}(i)$. Then we must update our various vectors. We perform $I.insert_{\mathbf{1}}(i')$ on our insert vector, and $D.insert_{\mathbf{0}}(i')$ on our delete vector.

$\hat{T}.delete(i)$. First, we convert position i in $\hat{T}$ to its corresponding position i' in T' by computing $i' = D.select_{\mathbf{0}}(i)$. If i' is newly-inserted (i.e., $I[i'] = \mathbf{1}$), then we perform $I.delete(i')$ and $D.delete(i')$ to reverse the insertion process from above. Otherwise, we first convert position i' in T' to its corresponding position i'' in T by computing $i'' = I.rank_{\mathbf{0}}(i')$. Let $s = T.char(i'')$. Finally, to delete the symbol, we perform $D.flip(i')$ and $D_s.flip(j)$, where $j = T.rank_s(i'')$.

$\hat{T}.rank_s(i)$. First, we convert position i in $\hat{T}$ to its corresponding position i' in T' by computing $i' = D.select_{\mathbf{0}}(i)$. If $s = \texttt{x}$, return $I.rank_{\mathbf{1}}(i')$. Otherwise, we first convert position i' in T' to its corresponding position i'' in T by computing $i'' = I.rank_{\mathbf{0}}(i')$. Finally, we return $D_s.rank_{\mathbf{0}}(j)$, where $j = T.rank_s(i'')$.

$\hat{T}.select_s(i)$. If $s = \texttt{x}$, compute $j = I.select_{\mathbf{1}}(i)$ and return $D.rank_{\mathbf{0}}(j)$. Otherwise, we compute $k = D_s.select_{\mathbf{0}}(i)$ to determine i's position among the s symbols from T. We then compute $k' = T.select_s(k)$ to determine its original position in T. Now the position k' from T needs to be mapped to its appropriate location in $\hat{T}$. Similar to the first case, we perform $k'' = I.select_{\mathbf{0}}(k')$ and return $D.rank_{\mathbf{0}}(k'')$, which corresponds to the right position of $\hat{T}$.

$\hat{T}.char(i)$. First, we convert position i in $\hat{T}$ to its corresponding position i' in T' by computing $i' = D.select_{\mathbf{0}}(i)$. If $I[i'] = 1$, return $\mathtt{x}$. Otherwise, we convert position i' in T' to its corresponding position i'' in T by computing $i'' = I.rank_{\mathbf{0}}(i')$ and return $T.char(i'')$.

Space and Time. As can be seen, each of the *rank* and *select* operations requires a constant number of accesses to BitIndel and StaticRankSelect structures, thus taking $O(1)$ time to perform. The *indel* operations require $O(n^\epsilon)$ update time, owing to the BitIndel data structure. The space required for the above data structures comes from the StaticRankSelect structure, which requires $s(n) = O(n \lg |\Sigma| + o(n \lg |\Sigma|))$ bits of space, and the many BitIndel structures, whose space can be bounded by $3 \lg \binom{n'}{n} + 6 \lg \binom{n'}{n''} + o(n') + O((n'/n^\epsilon) \lg n')$ bits where n'' is number of deletes. If n'' and $n' - n$ are bounded by $n^{1-\epsilon}$, then this expression is $o(n)$ bits.

Theorem 20. *Let T be a dynamic text of original length n and current length n', with characters drawn from an alphabet Σ. Let n'' be the number of deletions. If the number of updates is $O(n^{1-\epsilon})$, we can create a data structure using GMR that takes $n \lg |\Sigma| + o(n \lg |\Sigma|)$ bits of space and supports $rank_s(i)$ and $select_s(i)$ in $O(1)$ time and $insert_{\mathtt{x}}(i)$ and $delete_s(i)$ in $O(n^\epsilon)$ time.*

5.3.4 onlyX-structure

Let T be the dynamic text that we want to maintain, where symbols of T are drawn from alphabet Σ. Let n' be the current length of T, and we assume that $n' = O(n)$. In this section, we describe a data structure for maintaining a dynamic array of symbols that supports $rank_s$ and $select_s$ queries in $O((\lg_n n')(t(n) + \lg \lg n))$ time, for any fixed ϵ with

$0 < \epsilon < 1$; here, we assume that the maximum number of symbols in the array is $O(n)$. Our data structure takes $O(n' \lg n)$ bits; for each update (i.e., insertion or deletion of a symbol), it can be done in amortized $O(n^\epsilon)$ time.

We describe how to apply the WBB tree to maintain T while supporting $rank_s$ and $select_s$ efficiently, for any $s \in \Sigma$.[2] In particular, we choose $\epsilon < 1$ and store the symbols of T in a WBB W with fanout factor $b = n^\delta$ where $\delta = \epsilon/2$ such that the ith (leftmost) leaf of W stores $T[i]$. Each node at level 1 will correspond to a substring of T with $O(b)$ symbols, and we will maintain a static text dictionary for that substring so that $rank_s$ and $select_s$ are computed for that substring in $t(n) = O(\lg\lg|\Sigma|)$ time. In each level-ℓ node v_ℓ with $\ell \geq 2$, we store an array *size* such that *size*$[i]$ stores the number of symbols in the subtree of its ith (leftmost) child. To have fast access to this information at each node, we build a PS structure to store *size*. Also, for each symbol s that appears in the subtree of v_ℓ, v_ℓ is associated with an *s-structure*, which consists of three arrays: pos_s, num_s, and ptr_s. The entry $pos_s[i]$ stores the index of v_ℓ's ith leftmost child whose subtree contains s. The entry $num_s[i]$ stores the number of s in v_ℓ's ith leftmost child whose subtree contains s. The entry $ptr_s[i]$ stores a pointer to the s-structure of v_ℓ's ith leftmost child whose subtree contains s.

The arrays in each s-structure ($size_s$, pos_s, and num_s) are stored using a PS data structure so that we can support $O(\lg\lg n)$-time *sum* and *findsum* queries in $size_s$ or num_s, and $O(\lg\lg n)$-time *rank* and *select* queries in pos_s. (These *rank* and *select* operations are analogous to *sum* and *findsum* queries, but we refer to them as *rank* and *select* for ease of exposition.) The list ptr_s is stored in a simple array.

We also maintain another B-tree B with fanout n^δ such that each leaf ℓ_s corresponds to a symbol s that is currently present in the text T. Each leaf stores the number of (nonzero)

[2]One may think of using a B-tree instead of a WBB-tree. However, in our design, a particular node in the WBB tree will need to store auxiliary information about every symbol in the subtree under that node. In the worst case, this auxiliary information will be as big as the size of the subtree. If we use a B-tree, the cost of updating a particular node cannot bounded by $O(n^\epsilon)$ time in the amortized case.

occurrences of s in T, along with a pointer to its corresponding s-structure in the root of W. The height of B is $O(\lg_{n^\epsilon} |\Sigma|) = O(1)$, since we assume $|\Sigma| \leq n$.

Answering *char*(i). We can answer this query in $O(\lg \lg n)$ time by maintaining a B-tree with fanout $b = n^\delta$ over the text. We call this tree the *text B-tree*.

Answering *rank*$_s(p)$. Recall that $rank_s(p)$ records the number of occurrences of symbol s in $T[1..p]$. We first query B to determine if s occurs in T. If not, return 0. Otherwise, we follow the pointer from B to its s-structure. We then perform $r.size_s.findsum(p)$ to determine the child c_i of root r from W that contains $T[p]$. Suppose that $T[p]$ is in the subtree rooted at the ith child c_i of r. Then, $rank_s$ consists of two parts: the number of occurrences $m_1 = r.num_s.sum(j)$ (with $j = r.pos_s.rank(i-1)$) in the first $i-1$ children of r, and m_2, the number of occurrences of s in c_i. If $r.pos_s.rank(i) \neq j+1$ (c_i contains no s symbols), return m_1. Otherwise, we retrieve the s-structure of c_i by its pointer $r.ptr[j+1]$ and continue counting the remaining occurrences of s before $T[p]$ in the WBB tree W. We will eventually return $m_1 + m_2$.

The above process either `(i)` stops at some ancestor of the leaf of $T[p]$ whose subtree does not contain s, in which case we can report the desired rank, or `(ii)` it stops at the level-1 node containing $T[p]$, in which case the number of remaining occurrences can be determined by a $rank_s$ query in the static text dictionary in $t(n) = O(\lg \lg |\Sigma|)$ time. Since it takes $O(\lg \lg n)$ time to check the B-tree B at the beginning, and it takes $O(\lg \lg n)$ time to descend each of the $O(1)$ levels in the WBB-tree to count the remaining occurrences, the total time is $O(\lg \lg n)$.

Answering *select*$_s(j)$. Recall that $select_s(j)$ tells the number of symbols (inclusive) before the jth occurrence of s in T. We follow a similar procedure to the above procedure for $rank_s$. We first query B to determine if s occurs at least j times in T. If not, we return -1. Otherwise, we discover the ith child c_i of root r from W that contains the jth s symbol. We compute $i = r.pos_s.select(r.num_s.findsum(j))$ to find out c_i.

Then, select_s consists of two parts: the number of symbols $m_1 = r.\mathit{size}.\mathit{sum}(i)$ in the first $i-1$ children of r, and m_2, the number of symbols in c_i before the jth s. We retrieve the s-structure of c_i by its pointer $r.\mathit{ptr}[r.\mathit{num}_s.\mathit{findsum}(j)]$ and continue counting the remaining symbols on or before the jth occurrence of s in T. We will eventually return $m_1 + m_2$. The above process will stop at the level-1 node containing the jth occurrence of s in T, in which case the number of symbols on or before it maintained by this level-1 node can be determined by a select_s query in the static text dictionary in $t(n) = O(\lg\lg|\Sigma|)$ time. With similar time analysis as in rank_s, the total time is $O(\lg\lg n)$.

Updates. We can update the text B-tree in $O(n^\epsilon)$ time. We use a naive approach to handle updates due to the insertion or deletion of symbols in T: For each list in the WBB-tree and for each static text dictionary that is affected, we rebuild it from scratch. In the case that no split, merge, or merge-then-split operation occurs in the WBB-tree, an insertion or deletion of s at $T[p]$ will affect the static text dictionary containing $T[p]$, and two structures in each ancestor node of the leaf containing $T[p]$: the *size* array and the s-structure corresponding to the inserted (deleted) symbol. The update cost is $O(n^\delta \lg n) = O(n^\epsilon)$ for the static text dictionary and for each ancestor, so in total it takes $O(n^\epsilon)$ time.

If a split, merge, or merge-then-split operation occurs at some level-ℓ node v_ℓ in the WBB-tree, we need to rebuild the *size* array and s-structures for all newly created nodes, along with updating the *size* array and s-structures of the parent of v_ℓ. In the worst case, it requires $O(n^{(\ell+1)\epsilon} \lg n)$ time. By the property of WBB trees, the amortized update takes $O(n^\epsilon)$ time.

In summary, each update due to an insertion or deletion of symbols in T can be done in amortized $O(n^\epsilon)$ time.

Space complexity. The space for the text B-tree is $O(n\lg|\Sigma| + n^{1-\epsilon}\lg n)$ bits. The total space of all $O(n^{1-\epsilon})$ static text dictionaries can be bounded by $s(n) = O(n\lg|\Sigma|)$ bits.

For the space of the s-structures, it seems like it is $O(|\Sigma| n^{1-\epsilon} \lg n)$ bits at the first glance, since there are $O(n^{1-\epsilon})$ nodes in W. This space however is not desirable, since $|\Sigma|$

can be as large as n. In fact, a closer look of our design reveals that each node in W only maintains s-structures for those s that appears in its subtree. In total, each character of T contributes to at most $O(1)$ s-structures, thus incurring only $O(\lg n)$ bits. The total space for s structures is thus bounded by $O(n \lg n)$ bits.

The space for the B-tree B (maintaining distinct symbols in T) is $O(|\Sigma| \lg n)$ bits, which is at most $O(n \lg n)$ bits. In summary, the total space of the above dynamic rank-select structure is $O(n \lg n)$ bits.

Summarizing the above discussions, we arrive at the following theorem.

Theorem 21. *For a dynamic text T of length at most $O(n)$, we can maintain a data structure on T using GMR to support $rank_s$, $select_s$, and char $O(t(n)+\lg\lg n) = O(\lg\lg n)$ time, and insertion/deletion of a symbol in amortized $O(n^\epsilon)$ time. The space of the data structure is $O(n \lg n)$ bits.* $\square$

5.4 Constant-time onlyX-structure

For the case when $|\Sigma| = O(\text{polylg}(n))$, we can modify the onlyX structure so as to achieve $O(1)$ queries. This modification is similar to the one we made for our $O(1)$ BitIndel structure.

Precisely, let T be the dynamic text we want to maintain, n' be the length of T (which is never more than $2n$), and $\delta = \epsilon/2$ be a fixed constant. We maintain a WBB tree B for T to answer the $rank_s$ and *char* query, and a WBB tree V_s for each $s \in |\Sigma|$ to answer the corresponding $select_s$ query. For the WBB tree B, the fanout is $b = n^\delta$, so that each level-1 node corresponds to a block of $\Theta(b)$ characters of T. These characters are maintained by the StaticRankSelect structure of [NFMM06]. For each level-ℓ internal node v in the tree with $\ell \geq 2$, we define an array *size* such that *size*$[i]$ stores the number of characters in the subtree of its ith child, which is maintained by a PS structure of [HSS03]. We also store an array $count_s$ such that $count_s[i]$ stores the number of character s in the subtree of the ith child.

With the WBB tree B, $rank_s(i)$ can be answered by counting the number of s on or before the $T[i]$. This is done by (i) traversing B from root to the level-1 node v containing $T[i]$ based on the PS structures, and summing up the corresponding $count_s$ along the way, and then (ii) querying the StaticRankSelect structure of v for the remaining counts. The height of the tree is $O(1)$ and each level can be traversed in $O(1)$ time, $rank_s(i)$ is answered in $O(1)$ time. Similarly, we can use B to answer $char(i)$ query in $O(1)$ time.

For the WBB tree V_s for answer $select_s$ query, we use a similar approach as we define the Constant Time BitIndel structure. The weight is now balanced on $count_s$ (the number of s in the subtree), instead of *size* (the number of characters in the subtree). Each level-1 node will correspond to $\Theta(b)$ s, and depending on the sparsity of these characters, they will either be stored explicitly (if the position of the last s is at least $b \lg n$ characters away from the position of the first s), or will be considered as a bitvector and stored by a RRR structure. For the level-ℓ nodes with $\ell \geq 2$, we define the array $count_s$ such that $count_s[i]$ stores the number of s in the subtree of its ith child, which is maintained by a PS structure of [HSS03]. We also store an array *size* such that *size* stores the number of characters in the subtree of the ith child. With the WBB tree V_s, $select_s(i)$ can be answered by counting the number of characters before the ith s. This is done by (i) traversing V_s from root to the level-1 node v containing the ith s based on the PS structures, and summing up the corresponding *size* along the way, and then (ii) querying the explicit array or the RRR structure of v to count the remaining characters before the ith s. The height of the tree is $O(1)$ and each level can be traversed in $O(1)$ time, $select_s(i)$ is answered in $O(1)$ time.

The total space of the data structure is bounded by $O(|\Sigma| n \lg n)$ bits. For updating due to insertion or deletion of a character, it is again performed by a naive approach—rebuild the affected nodes from scratch. The amortized update time can be easily bounded by $O(b|\Sigma| \lg^2 n) = O(n^\epsilon)$. And for the working space to perform the updates, observe that we can fix each node of each WBB tree one by one. Thus, the working space is only $O(b \lg n) = O(n^\epsilon)$ bits.

Summarizing, we have the following theorem.

Theorem 22. *Suppose that $|\Sigma| = \text{polylg}(n)$. For a dynamic text T of length at most $O(n)$, we can maintain a data structure on T using the wavelet tree to support $rank_s$, $select_s$, and char in $O(t(n)) = O(1)$ time, and insertion/deletion of a symbol in amortized $O(n^\epsilon)$ time. The space of the data structure is $O(|\Sigma| n \lg n)$ bits, and the working space to perform the updates at any time is $O(n^\epsilon)$ bits.* □

5.4.1 The Final Data Structure

Here we describe our final structure, which supports insertions and deletions of any symbol. To do this, we maintain two structures: our inX structure on $\hat{T}$ and the onlyX structure, where all of the new symbols are actually inserted and maintained. After every $O(n^{1-\epsilon} \lg n)$ update operations, the onlyX structure is merged into the original text T and a new T is generated. All associated data structures are also rebuilt. Since this construction process could take at most $O(n \lg n)$ time, this cost can be amortized to $O(n^\epsilon)$ per update. The StaticRankSelect structure on T takes $s(n) = n \lg |\Sigma| + o(n \lg |\Sigma|)$ bits of space. With this frequent rebuilding, all of the other supporting structures take only $o(n)$ bits of space.

We augment the above two structures with a few additional BitIndel structures. In particular, for each symbol s, we maintain a bitvector I_s such that $I_s[i] = \mathbf{1}$ if and only if the ith occurrence of s is stored in the onlyX structure. With the above structures, we quickly describe how to support $rank_s(i)$ and $select_s(i)$.

For $rank_s(i)$, we first find $j = inX.rank_s(i)$. We then find $k = inX.rank_{\mathtt{x}}(i)$ and return $j + onlyX.rank_s(k)$. For $select_s(i)$, we first find whether the ith occurrence of c belongs to the inX structure or the onlyX structure. If $I_s[i] = \mathbf{0}$, this means that the ith item is one of the original symbols from T; we query $inX.select_s(j)$ in this case, where $j = I_s.rank_{\mathbf{0}}(i)$. Otherwise, we compute $j = I_s.rank_{\mathbf{1}}(i)$ to translate i into its corresponding position among new symbols. Then, we compute $j' = onlyX.select_s(j)$, its location in $\hat{T}$ and return $inX.select_x(j')$.

Finally, we show how to maintain I_s during updates. For $delete(i)$, compute $\hat{T}[i] = s$. We then perform $I_s.delete(inX.rank_s(i))$. For $insert_s(i)$, after inserting s in $\hat{T}$, we insert it into I_s by performing $I_s.insert_{\mathbf{1}}(inX.rank_s(i))$. Let $n_{\mathtt{x}}$ be the number of symbols stored in the onlyX structure. We can bound the space for these new BitIndel data structures using RRR [RRR02] and Jensen's inequality by $\lceil \lg \binom{n'}{n_{\mathtt{x}}} \rceil + o(n') = O(n^{1-\epsilon} \lg^2 n) + o(n) = o(n)$ bits of space. Thus, we arrive at the following theorem.

Theorem 23. *Given a text T of length n drawn from an alphabet Σ, we create a data structure using GMR that takes $s(n) = n \lg |\Sigma| + o(n \lg |\Sigma|) + o(n)$ bits of space and supports $rank_s(i)$, $select_s(i)$, and $char(i)$ in $O(\lg \lg n + t(n)) = O(\lg \lg n + \lg \lg |\Sigma|)$ time and $insert(i)$ and $delete(i)$ updates in $O(n^{\epsilon})$ time.* □

For the special case when $|\Sigma| = \text{polylg}(n)$, we may now use [NFMM06] as the StaticRankSelect structure, and the Constant Time BitIndel as the BitIndel structure. For the onlyX structure, we use the one described in Section 5.4, whose space is $o(n)$ if merging is performed every $O(n^{1-\epsilon})$ update operations. Then, we achieve the following theorem.

Theorem 24. *Given a text T of length n drawn from an alphabet Σ, with $|\Sigma| = \text{polylg}(n)$, we create a data structure using the wavelet tree that takes $s(n) + o(n) = nH_0 + o(n \lg |\Sigma|) + o(n)$ bits of space and supports $rank_s(i)$, $select_s(i)$, and $char(i)$ in $O(t(n)) = O(1)$ time and $insert(i)$ and $delete(i)$ updates in $O(n^{\epsilon})$ time.* □

We skip the details about the memory allocation issues for our dynamic structures and rebuilding space issues. However, the overhead for these issues can be shown to be $o(n)$ bits of additional space.

5.5 Dynamizing Ordinal Trees, Labeled Trees, and the XBW Transform

In this section, we describe applications of our BitIndel data structure and our dynamic multi-symbol rank/select data structure to dynamizing ordinal trees, labeled trees, and the XBW transform [FLMM05].

Ordinal Trees. An *ordinal tree* is a rooted tree where the children are ordered and specified by their rank. An ordinal tree can be represented by the Jacobson's LOUDS representation [BDM⁺05] using just *rank* and *select*. Thus, we can use our BitIndel data structure to represent any ordinal tree with the following operations:

- $v.parent()$, returns the parent node of v in T;
- $v.child(i)$, returns the ith child node of v;
- $v.insert(k)$, inserts the kth child of node v;
- $v.delete(k)$, removes the kth child of node v;

Lemma 41. *For any ordinal tree T with n nodes, there exists a dynamic representation of it that takes at most $2n + O(n \lg\lg n/\lg n)$ bits of space and supports updates in amortized $O(n^\epsilon)$ time and navigational queries in $O(\lg\lg n)$ time. Alternatively, we can take $6n + O(n \lg\lg n/\lg n)$ bits of space and support navigational queries in just $O(1)$ time.* □

Labeled Trees, Text Collections, and XBW. A labeled tree T is a tree where each of the n nodes is associated with a label from alphabet Σ. To ease our notation, we will also number our symbols from $[0, |\Sigma| - 1]$ such that the sth symbol is also the sth lexicographically-ordered one. We'll call this symbol s. We are interested in constructing a data structure that supports the following operations in T:

- $insert(P)$, inserts the path P into T;
- $v.delete()$, removes the root-to-v path for a leaf v;
- $subpath(P)$, finds all occurrences of the path P;
- $v.parent()$, returns the parent node of v in T;
- $v.child(i)$, returns the ith child node of v; and
- $v.child(s)$, returns any child node of v labeled s.

Ferragina et al. [FLMM05] propose an elegant way to solve the static version of this problem by performing an XBW transform on the tree T, which produces an XBW text S. They show that storing S is sufficient to support the desired operations on T efficiently, namely navigational queries in $O(\lg |\Sigma|)$ time and $subpath(P)$ queries in $O(|P| \lg |\Sigma|)$ time.

In the dynamic case when we want to support insert or delete of a path of length m, we observe that either operation corresponds to an update of this XBW text S at m positions. Using our dynamic framework, we can maintain a dynamic version of this text S and achieve similar results for the dynamic case.

Before explaining our data structure, we first give a brief description of the XBW transform [FLMM05]. For a node v in T, let $\ell[v] = \mathbf{1}$ if and only if v is the rightmost child of its parent in T. Let $\alpha[v]$ be the label of v, and $\pi[v]$ be the string obtained by concatenating the labels on the upward path from $v.parent()$ to the root of T. We further assume that the node labels can be separated into two disjoint sets Σ_i and Σ_l of labels for internal nodes and leaves (respectively). We also let n_i be the number of internal nodes of T and n_ℓ be the number of leaves of T. We then construct a set S of n triplets, one for each tree node:

- Visit T in pre-order. For each visited node v add the triplet $s[v] = \langle \ell[v], \alpha[v], \pi[v] \rangle$ into S;
- Stable-sort S according to the π component of each triple.

The (output of the) XBW transform consists of the arrays S_ℓ and S_α, where these refer to the first and second components of each triplet (respectively) after the stable sort has been performed. Ferragina, et al show in [NFMM06] that the tree T can be reconstructed by storing these arrays. The above transform is reminiscent of the Burrows-Wheeler Transform (BWT) for text documents. Their structure supports navigational queries (*parent*, *child*) operations, as well as a *subpath*(P) search, which finds the nodes v such that the reversed path $rev(P)$ is a prefix of the concatenated string $\alpha[v]\pi[v]$. In summary, they achieve the following theorem for the static ordered trees T:

Theorem 25 (Static XBW [FLMM05]). *For any ordered tree T with node labels drawn from an alphabet Σ, there exists a static succinct representation of it using the XBW transform that takes at most $nH_0(S_\alpha) + 2n + o(n)$ bits of space, while supporting navigational queries in $O(\lg |\Sigma|)$ time. The representation can also answer a subpath(P) query in $O(m \lg |\Sigma|)$ time, where m is the length of path P.* $\square$

The full details of the result can be found in [FLMM05]. Here, we briefly recap the data structures used in their solution. For our result, we will show that replacing these structures with their dynamic counterpart is sufficient to achieve a powerful facility to update ordered trees (such as XML trees). For S_ℓ, [FLMM05] use an RRR [RRR02] data structure to maintain the bitvector of length n containing n_i **1**s in $\lg \binom{n}{n_i} + o(n)$ bits of space. For S_α, [FLMM05] keep two data structures: F and S_α. The data structure F keeps track of the number of occurences of each symbol s in Σ. F is (conceptually) a bitvector of length $n + |\Sigma|$ storing $|\Sigma|$ **1**s such that $\mathit{select}_{\mathbf{1}}(i) - \mathit{select}_{\mathbf{1}}(i-1) - 1$ indicates the number of occurrences of the ith label s in T. Finally, S_α is stored using a wavelet tree [GGV03].

For our dynamic XBW data structure, we replace the static implementations of S_ℓ and F with our BitIndel data structure, supporting *rank* and *select* in $O(\lg\lg n)$ time and updates in $O(\lg_n n' + n^\epsilon)$ amortized time. Then, we replace the S_α data structure with our "final structure" that allows rank_s and select_s in $O(\lg\lg n)$ time and supports insertions and deletions in $O(n^\epsilon)$ time. We use the same algorithms for *parent* and *child* operations as [FLMM05]. Since these algorithms require a constant number of queries to the above data structures, we can now support these operations in $O(\lg\lg n)$ time. For $\mathit{subpath}(P)$, we again use the same algorithm, taking $O(m \lg\lg n)$ time, where m is the length of P.

For $\mathit{insert}(P)$ and $\mathit{delete}()$, these operations will be defined on the original tree T for some node u where we want to begin inserting or deleting. We describe a method to translate any node u into a corresponding position v such that the triplet $S[v]$ in the XBW transform [FLMM05] corresponds to node u in T. For a path from root r to a node u in T, say $P = (u_0, u_1, u_2, \cdots, u_{h-1}, u_h)$ with $u_0 = r$ and $u_h = u$, we describe a sequence of child indices $C_u = c_1 c_2 \ldots c_h$, where c_i indicates that u_i is the c_ith child of u_{i-1}. To translate u into the corresponding position v in the XBW transform [FLMM05], we perform the following *convert* operation.

```
function $convert(C_u)$ {
  $v \leftarrow 1$; // $v$ is the root
  for $(i = 1; i \leq h; i{+}{+})$
    $v \leftarrow v.child(c_i)$;
  return $v$;
}
```

The above operation takes $O(h \lg\lg n)$ time to perform with our dynamic data structures, where $h + 1$ is the depth of the node to be modified. Our later operations will take this much additional time. We state the following lemma.

Lemma 42. *For any node u at depth $h + 1$ in tree T, we can find its corresponding position in the XBW transform [FLMM05] in $O(h \cdot t(n))$ time, where $t(n)$ is the amount of time to perform a $child(i)$ navigational operation by a data structure storing the XBW transform.* □

We now describe how to support *insert*(P) and $v.delete()$ for node v in the XBW transform [FLMM05]. For convenience, we rewrite $P = p_1 p_2 \cdots p_m$ as the concatenation of its m symbols. Furthermore, we assume that node v refers to its position in the XBW transform (easily done with *convert*(c_v)). For *insert*(P), we traverse the path P in the XBW transform until we encounter a leaf v. We find v's last child. We then insert the next symbol in P after this child, making the appropriate changes to S_ℓ and S_α. We also update F so that it maintains the correct count of alphabet symbols. For $v.delete()$, note that it's sufficient to simply know the leaf node $l = v$ of the path we wish to delete. To execute a deletion, we remove this leaf l and propagate to l's parent, making the appropriate changes to F, S_ℓ, and S_α. We terminate if l's parent has more than one child. We show the pseudo-code below. (We assume we can access the value of any entry stored in the data structures by our previous discussion.)

```
function $v.insert(p_1 p_2 \cdots p_m)$ {
  if ($S_\alpha[v] \in \Sigma_l$) return $-1$;
  $s \leftarrow S_\alpha[v]$;
  $y \leftarrow F.select_{\mathbf{1}}(s)$;
  $k \leftarrow S_\alpha.rank_s(v)$;
  $z \leftarrow S_\ell.rank_{\mathbf{1}}(y-1)$;
  $v' \leftarrow S_\ell.select_{\mathbf{1}}(z+k)$;
  $S_\ell.flip(v')$;
  $S_\ell.insert_{\mathbf{1}}(v'+1)$;
  $S_\alpha.insert_{p_1}(v'+1)$;
  $F.insert_{\mathbf{1}}(F.select_{\mathbf{1}}(p_1)+1)$;
  $(v'+1).insert(p_2 \cdots p_m)$;
}
```

```
function $v.delete()$ { // $v$ has no children
  $s \leftarrow S_\alpha[v]$;
  $y \leftarrow F.select_{\mathbf{1}}(s)$;
  $k \leftarrow S_\ell.rank_{\mathbf{1}}(v-1) - S_\ell.rank_{\mathbf{1}}(y-1)$;
  $p \leftarrow S_\alpha.select_s(k+1)$;
  $F.delete(F.select_{\mathbf{1}}(s)+1)$;
  $S_\alpha.delete(v)$;
  if ($S_\ell[v] = \mathbf{0}$)
    $S_\ell.delete(v)$;
    exit;
  else if ($S_\ell[v-1] = \mathbf{0}$)
    $S_\ell.flip(v-1)$;
    $S_\ell.delete(v)$;
    exit;
  if ($p < v$)
    $p.delete()$;
  else
    $(p-1).delete()$;
}
```

The above process can be expanded to also include routines for subtree insertion and deletion (*tinsert*, *tdelete*). Notice that the above algorithms require $O(m)$ queries to our dynamic data structures to insert or delete a path of length m. Thus, we arrive at the following theorem using GMR.

Theorem 26 (Dynamic XBW). *For any ordered tree T, there exists a dynamic succinct representation of it using the XBW transform [FLMM05] that takes at most $s(n) + 2n = n \lg |\Sigma| + o(n \lg |\Sigma|) + 2n$ bits of space, while supporting navigational queries in $O(t(n) + \lg\lg n) = O(\lg\lg n)$ time. The representation can also answer a subpath(P) query in $O(m(t(n) + \lg\lg n)) = O(m \lg\lg n)$ time, where m is the length of path P. The update* *operations insert*(P) *and delete*$()$ *at node* u *for this structure take* $O(n^\epsilon + m(t(n) + \lg\lg n))$ *amortized time, where* m *is the length of the path* P *being inserted or deleted.* □

Chapter 6

Conclusions and Future Directions

In this thesis, we have explored the notion of compressing data while retaining its accessibility for important queries in competitive time bounds. From general text indexing to various instances of dictionary problems, succinct data structures can serve as replacements for their corresponding non-succinct versions without a significant tradeoff in query performance. In theory, a more ubiquitous use of these data structures seems like a natural progression. In a practical setting, we have discovered time and again that these succinct data structures really *can* make a difference in storing the data. Real-life data rarely exhibits worst-case or random behavior, so our measures and techniques truly do reduce the data stored.

Our work is just the tip of othe iceberg. By itself, compression can lead to insights in understanding the underlying structure or information in a large amount of data, possibly even a data set that contains a lot of "noise"; it can reduce network load [AAG$^+$95, GKKV95], I/O overhead [Vit01], or save battery power on mobile devices. Compression techniques can also be used as a tool to predict future trends and behavior [CKV93, KV98]. Paired with fast query access, we can apply these goals to a wide variety of problems and expand the power of queries that we consider. To this end, we encourage researchers to develop theoretically and practically succinct data structures using a data-aware analysis. We briefly mention a few possible directions where these themes can be expanded and explored.

IP Lookup Problem. Computer networks are expected to exhibit very high performance in delivering data, owing to the explosive growth of Internet nodes. Routers forward many packets from input to output interfaces, based on the destination address of the packet. Briefly, forwarding a packet requires an IP address lookup in a routing table to select the next hop appropriate for the packet. Because of the bottleneck on computation time available to the router, this simple IP lookup is practically prohibitive. With such a realization, early assumptions of the ease of IP lookups have vanished, replaced by the reality that it is inconceivable to store all existing IP addresses explicitly, since routing tables would contain millions of entries. In terms of our dictionary structures, given a query IP address (as a string), our task would be to find the item in our dictionary (composed of a subset of all possible IP addresses) having the *longest prefix match* with the query address. The challenge is to develop a sound theoretical structure that is simple enough to provide blazingly fast practical results, while still retaining space efficiency.

Text Indexing. A basic open problem remains in how to make compressed suffix arrays (and in general, text indexes) dynamic; another question is whether it is possible for the CSA to be I/O efficient [Vit01]. Many applications appear in Gusfield's book [Gus97a] that use suffix arrays, suffix trees, and their variants. For instance, we highlight a few examples (many relevant to applications in computational biology), such as the space-efficient longest common substring problem, finding all maximal palindromes in linear time, exact matching with wildcards, the k-mismatch problem, among others.

Multidimensional Matching. An interesting extension of our text indexing work, with practical applications related to image matching, is to develop a data structure that achieves similar space bounds as the 1-D case and the same time bounds as known multidimensional data structures. Multidimensional data present a new challenge when trying to capture entropy, as now the critical notion of *spatial information* also enters into play. (In a strict sense, this information was always present, but we can anticipate more dependence upon spatially linked data.) Stronger notions of compression are applicable, yet the searches are more complicated. Achieving both, is again, a challenge.

Approximate Matching. Another major series of extensions to our text indexing work deals with improving the quality of the search functionality provided. The two major flavors of search functionality are fault-tolerant (approximate) matches and wild-card matches. Wild-card matches are a subset of (and thus easier than) approximate matches. Generally speaking, approximate matching is of a great deal of interest to a number of communities. Computational biologists want to find “related objects” in their searches [Gus97a], without being constrained to the strict notion of exactness. Inspecting audio, video, or image clips for patterns rarely demand exact matches.

There has been a lot of work on approximate matching, especially in the computational biology community. A comprehensive survey by Navarro [Nav01] provides insights on the issues involved. While edit distance (LCS measure) is one of the most popular approximation criteria, many others (like hamming distance, metric distance, etc.[MS00, MS02]) have been considered as well. In spite of considerable progress in approximate pattern matching, there has been very little positive development on indexed searching for approximate matches. The known index structures for approximate matching tend to take a huge amount of space, many times the text size. Indexed approximate searching is a difficult problem and the area is quite new and active. There have been some recent results by Navarro et al. [MNZBY98, NBY00, NBYST01].

Bibliography

[AAG$^+$95] B. Awerbuch, Y. Azar, E. F. Grove, M. Y. Kao, P. Krishnan, and J. S. Vitter. Load balancing in the l_p norm. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, volume 36, pages 383–391, October 1995.

[Aar05] Scott Aaronson. NP-complete problems and physical reality. *SIGACT News*, 36(1):30, 2005.

[AASA01] Hiroki Arimura, Hiroki Asaka, Hiroshi Sakamoto, and Setsuo Arikawa. Efficient discovery of proximity patterns with suffix arrays (extended abstract). In *CPM: 12th Symposium on Combinatorial Pattern Matching*, 2001.

[AKO04] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[AT00] Arne Andersson and Mikkel Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *ACM Symposium on Theory of Computing (STOC)*, 2000.

[AUT] `http://ccrma-www.stanford.edu/~jos/mdft/Autocorrelation.html`.

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[Bau04] Eric Baum. *What is Thought?* MIT Press, 2004.

[BB04] Daniel K. Blandford and Guy E. Blelloch. Compact representations of ordered sets. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, January 2004.

[BB05] Daniel K. Blandford and Guy E. Blelloch. Dictionaries using variable-length keys and data, with applications. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, January 2005.

[BBK03] Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. Compact representations of separable graphs. pages 679–688, 2003.

[BDFC05] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. *SIAM J. Comput.*, 2005. (Also in IEEE FOCS 2000.).

[BDM$^+$05] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[BF99] Paul Beame and Faith Fich. Optimal bounds for the predecessor problem. In *ACM Symposium on Theory of Computing (STOC)*, pages 295–304, 1999.

[BFC04] Michael A. Bender and Martin Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004.

[BM99] Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM Journal on Computing*, 28(5):1627–1640, October 1999.

[BMNM$^+$93] Timothy C. Bell, Alistair Moffat, Craig G. Nevill-Manning, Ian H. Witten, and Justin Zobel. Data compression in full-text retrieval systems. *Journal of the American Society for Information Science*, 44(9):508–531, 1993.

[BSTW86] Jon Bentley, Daniel Sleator, Robert Tarjan, and Victor Wei. A locally adaptive data compression scheme. *Communications of the ACM*, pages 320–330, 1986.

[BW94] M. Burrows and D.J. Wheeler. A block sorting data compression algorithm. Technical report, Digital Systems Research Center, 1994.

[Can] The Canterbury Corpus, `http://corpus.canterbury.ac.nz`.

[CDG99] Pierluigi Crescenzi, Leandro Dardini, and Roberto Grossi. Ip address lookup made fast and simple. In *European Symposium on Algorithms (ESA)*, pages 65–76, 1999.

[CG86] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

[Cha04] Bernard Chazelle. Who says you have to look at the input? The brave new world of sublinear computing, 2004. Plenary talk at at *the 15th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA 2004).

[CKV93] K. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 257–266, May 1993.

[CT91] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory.* Wiley-Interscience, New York, 1991.

[Deo02] Sebastian Deorowicz. Second step algorithms in the burrows-wheeler compression algorithm. In *Software–Practice and Experience*, volume 32, pages 99–111, 2002.

[DLO03] Erik D. Demaine and Alejandro López-Ortiz. A linear lower bound on index size for text retrieval. *J. Algorithms*, 48(1):2–15, 2003.

[Eli75] Peter Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21:194–203, 1975.

[EVKV02] Michelle Effros, Karthik Visweswariah, Sanjeev R. Kulkarni, and Sergio Verdu. Universal lossless source coding with the burrows-wheeler transform. *IEEE Transactions on Information Theory*, 48(5):1061–1081, 2002.

[Fel68] William Feller. *An Introduction to Probability Theory and its Applications*, volume 1. John Wiley & Sons, New York, 3rd edition, 1968.

[Fen96] Peter Fenwick. Punctured elias codes for variable-length coding of the integers. 1996. The University of Auckland, NZ. TR 137. ISSN 1173-3500.

[Fen02] Peter Fenwick. Burrows-Wheeler compression with variable-length integer codes. In *Software–Practice and Experience*, volume 32, pages 1307–1316, 2002.

[Fer92] David E. Ferguson. Bit-Tree: a data structure for fast file processing. *Communications of the ACM*, 35(6):114–120, June 1992.

[FGGV04] Luca Foschini, Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. Fast compression with a static model in high-order entropy. In *Proceedings of the IEEE Data Compression Conference*, Snowbird, UT, March 2004.

[FGMS05] Paolo Ferragina, Raffaele Giancarlo, Giovanni Manzini, and Gabriella Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, 52(4):688–713, 2005. (Also in CPM 2003, ACM-SIAM SODA 2004.).

[FLMM05] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 184–196, 2005.

[FM01] Paolo Ferragina and Giovanni Manzini. An experimental study of an opportunistic index. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 269–278. ACM/SIAM, 2001.

[FM05] Paolo Ferragina and Giovanni Manzini. On compressing and indexing data. *Journal of the ACM*, 52(4):552–581, 2005. (Also in IEEE FOCS 2000.).

[FMMN04] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Succinct representation of sequences. Technical Report DCC-2004-5, Departamento de Ciencias de la Computación, Universidad de Chile, August 2004. (Also in SPIRE 2004.).

[FTL03] Peter Fenwick, Mark Titchener, and Michelle Lorenz. Burrows Wheeler – alternatives to move to front. *Data Compression Conference (DCC)*, 2003.

[FW93] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

[GBS92] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. New indices for text: PAT trees and PAT arrays. In *Information Retrieval: Data Structures And Algorithms*, chapter 5, pages 66–82. Prentice-Hall, 1992.

[GGV03] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, January 2003.

[GGV04] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. January 2004.

[GK81] Daniel H. Greene and Donald E. Knuth. *Mathematics for the Analysis of Algorithms.* Birkhäuser, Boston, 1981.

[GKKV95] E. F. Grove, M. Y. Kao, P. Krishnan, and J. S. Vitter. Online perfect matching and mobile computing. In *Proceedings of the Workshop on Algorithms and Data Structures*, volume 955, pages 194–205, 1995.

[GM03] Anna Gál and Peter Bro Miltersen. The cell probe complexity of succinct data structures. In *Automata, Languages and Programming, 30th International Colloquium (ICALP 2003)*, volume 2719 of *Lecture Notes in Computer Science*, pages 332–344. Springer-Verlag, 2003.

[GMR06] Alexander Golynski, J. Ian Munro, and Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *SODA*, pages 368–373, 2006.

[GRR04] Richard F. Geary, Rajeev Raman, and Venkatesh Raman. Succinct ordinal trees with level-ancestor queries. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1–10. Society for Industrial and Applied Mathematics, 2004.

[Gus97a] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, Cambridge, UK, 1997.

[Gus97b] Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[GV00] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the ACM Symposium on Theory of Computing*, volume 32, May 2000.

[GV05] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[Hir78] Daniel S. Hirschberg. A lower worst-case complexity for searching a dictionary. In *Proc. 16th Annual Allerton Conference on Communication, Control, and Computing*, pages 50–53, 1978.

[HLS$^+$04] Wing-Kai Hon, Tak Wah Lam, Wing-Kin Sun, Wai-Leuk Tse, Chi-Kwong Wong, and Siu-Ming Yiu. Practical aspects of compressed suffix arrays and fm-index in searching dna sequences. In *6th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2004.

[HMP01] Torben Hagerup, Peter Bro Miltersen, and Rasmus Pagh. Deterministic dictionaries. 41(1):353–363, 2001.

[How97] Paul G. Howard. Interleaving entropy codes. In *Sequences*, 1997.

[HSS03] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Succinct data structures for searchable partial sums. In *ISAAC*, pages 505–516, 2003.

[HV94] Paul G. Howard and Jeffrey Scott Vitter. Arithmetic coding for data compression. *Proceedings of the IEEE*, 82(6), June 1994.

[Jac89a] Guy Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 549–554, 1989.

[Jac89b] Guy Jacobson. Succinct static data structures. Technical Report CMU-CS-89-112, Dept. of Computer Science, Carnegie-Mellon University, January 1989.

[KLA$^+$01] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa1, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Combinatorial Pattern Matching (CPM)*, pages 181–192, 2001.

[KLV06] Haim Kaplan, Shir Landau, and Elad Verbin. A simpler analysis of burrows-wheeler based compression. pages 282–293, 2006.

[KM99] S. Rao Kosaraju and Giovanni Manzini. Compression of low entropy strings with lempel-ziv algorithms. *SIAM J. Comput.*, 29(3):893–911, 1999.

[Knu05] Donald E. Knuth. *Combinatorial Algorithms*, volume 4 of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, USA, 2005. In preparation.

[KS02] Shmuel T. Klein and Dana Shapira. Searching in compressed dictionaries. In *Data Compression Conference (DCC)*, 2002.

[Kur99] Stefan Kurtz. Reducing the Space Requirement of Suffix Trees. *Software – Practice and Experience*, 29(13):1149–1171, 1999.

[KV98] P. Krishnan and Jeffrey Scott Vitter. Optimal prediction for prefetching in the worst case. *SIAM Journal on Computing*, 27(6):1617–1636, December 1998.

[lha] `http://www.infor.kanazawa-it.ac.jp/ ishii/lhaunix/`.

[LS97] Tomasz Luczak and Wojciech Szpankowski. A suboptimal lossy data compression based in approximate pattern matching. *IEEE Trans. Information Theory*, 43:1439–1451, 1997.

[LV97] Ming Li and Paul Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications.* Springer Verlag, 1997.

[Man01] Giovanni Manzini. An analysis of the Burrows — Wheeler transform. *Journal of the ACM*, 48(3):407–430, May 2001.

[McC76] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[Mil05] Peter Bro Miltersen. Lower bounds on the size of selection and rank indexes. In *Proc. the Sixteenth ACM-SIAM symposium on Discrete Algorithms (SODA05)*, pages 11–12, Philadelphia, PA, USA, 2005.

[MM93] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[MN06] Veli Mäkinen and Gonzalo Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 2006.

[MNW98] Alistair Moffat, Radford M. Neal, and Ian H. Witten. Arithmetic coding revisited. *ACM Transactions on Information Systems (TOIS)*, 16(3):256–294, 1998.

[MNZBY98] Edleno Moura, Gonzalo Navarro, Nivio Ziviani, and Ricardo Baeza-Yates. Fast searching on compressed text allowing errors. In B. Croft, A. Moffat, C. Rijsbergen, R. Wilkinson, and J. Zobel, editors, *Proceedings of the 21th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'98)*, pages 298–306. York Press, 1998.

[Mor68] Donald R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded In Alphanumeric. *Journal of the ACM*, 15(4):514–534, October 1968.

[MR99] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses, static trees, and planar graphs. *SIAM Journal on Computing*, 31:762–776, 1999.

[MR02] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, June 2002.

[MR04] J. Ian Munro and S. Srinivasa Rao. Succinct representations of functions. In *Annual International Colloquium on Automata, Languages and Programming (CALP)*, volume 3142 of *Lecture Notes in Computer Science*, pages 1006–1015. Springer-Verlag, 2004.

[MRRR03] J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct representations of permutations. In *Annual International Colloquium on Automata, Languages and Programming (CALP)*, volume 2719 of *Lecture Notes in Computer Science*, pages 345–356. Springer-Verlag, 2003.

[MRS01a] J. Ian Munro, Venkatesh Raman, and S. Srinivasa Rao. Space efficient suffix trees. *Journal of Algorithms*, 39:205–222, 2001.

[MRS01b] J. Ian Munro, Venkatesh Raman, and Adam J. Storm. Representing dynamic binary trees succinctly. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-01)*, pages 529–536, New York, January 7–9 2001. ACM Press.

[MS00] S. Muthukrishnan and Suleyman Cenk Sahinalp. Approximate nearest neighbors and sequence comparison with block operations. In *ACM Symposium on Theory of Computing (STOC)*, pages 416–424, 2000.

[MS02] S. Muthukrishnan and Suleyman Cenk Sahinalp. Simple and practical sequence nearest neighbors with block operations. In *Combinatorial Patteren Matching (CPM)*, pages 262–278, 2002.

[Mun96] J. Ian Munro. Tables. *FSTTCS: Foundations of Software Technology and Theoretical Computer Science*, 16:37–42, 1996.

[Mut03] S. Muthukrishnan. Data streams: Algorithms and applications, 2003. Plenary talk at *the 14th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA 2003).

[Nav01] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[NBY00] Gonzalo Navarro and Ricardo Baeza-Yates. A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms (JDA)*, 1(1):205–239, 2000. Special issue on Matching Patterns.

[NBYST01] Gonzalo Navarro, Ricardo Baeza-Yates, Erikki Sutinen, and Jose Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24(4):19–27, 2001. Special issue on Managing Text Natively and in DBMSs. Invited paper.

[Nel] Mark Nelson. Run length encoding/RLE. `http://www.datacompression.info/RLE.shtml`.

[NFMM06] Gonzalo Navarro, Paolo Ferragina, Giovanni Manzini, and Veli Mäkinen. Succinct representation of sequences and full-text indexes. *TALG*, 2006. To appear.

[NM06a] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. Technical Report TR/DCC-2006-6, University of Chile, 2006.

[NM06b] Gonzalo Navarro and Veli Mäkinen. Dynamic entropy-compressed sequences and full-text indexes. In *CPM*, pages 306–317, 2006.

[Pag99] Rasmus Pagh. Low redundancy in static dictionaries with $O(1)$ worst case lookup time. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*, volume 1644 of *Lecture Notes in Computer Science*, pages 595–604. Springer-Verlag, 1999.

[Pag01] Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31:353–363, 2001.

[PD06] Mihai Patrascu and Erik Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006.

[PT06] Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for predecessor search. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 232–240, 2006.

[Rao02] S. Srinivasa Rao. Time-space trade-offs for compressed suffix arrays. *IPL*, 82(6):307–311, 2002.

[RC93] John H. Reif and Shenfeng Chen. Using difficulty of prediction to decrease computation: Fast sort, priority queue and convex hull on entropy bounded inputs. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, volume 34, Palo Alto, 1993.

[Ris84] Jorma Rissanen. Universal coding, information, prediction, and estimation. *IEEE Transactions on Information Theory*, IT-30:629–636, 1984.

[RL79] Jorma Rissanen and Glen G. Langdon. Arithmetic coding. *IBM J. Research and Development*, 23(2):149–162, March 1979.

[RR03] Rajeev Raman and S. Srinivasa Rao. Succinct dynamic dictionaries and trees. In *Annual International Colloquium on Automata, Languages and Programming (CALP)*, volume 2719 of *Lecture Notes in Computer Science*, pages 357–368. Springer-Verlag, 2003.

[RRR01] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao. Succinct dynamic data structures. In *WADS*, pages 426–437, 2001.

[RRR02] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 233–242, 2002.

[Rus05] Frank Ruskey. *Combinatorial Generation.* 2005. In preparation.

[Sad02a] Kunihiko Sadakane, 2002. Personal Communication.

[Sad02b] Kunihiko Sadakane. Succinct representations of *lcp* information and improvements in the compressed suffix arrays. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms.* ACM/SIAM, 2002.

[Sad03] Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003. (Also in ISAAC 2000.).

[Sch] Michael Schindler. `http://www.compressconsult.com/rangecoder`.

[SG06] Kunihiko Sadakane and Roberto Grossi. Squeezing succinct data structures into entropy bounds. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1230–1239, 2006.

[Sha48] Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, July 1948.

[Tip] TREC Tipster 3. `http://trec.nist.gov/data/docs_eng.html`.

[Ukk95] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, September 1995.

[vEBKZ77] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Systems Theory*, 10:99–127, 1977.

[Vit84] Jeffrey Scott Vitter. Faster methods for random sampling. *Communications of the ACM*, 27(7):703–718, July 1984.

[Vit01] J. S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE DATA. *ACM Computing Surveys*, 33(2):209–271, June 2001. Revised version from August 2007 is also available at `http://www.cs.duke.edu/ jsv/Papers/catalog/node39.html`.

[VK96] Jeffrey Scott Vitter and P. Krishnan. Optimal prefetching via data compression. *Journal of the ACM*, 43(5), September 1996.

[Wei73] Peter Weiner. Linear pattern matching algorithm. *Proc. 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[Wil84] Dan E. Willard. New trie data structures which support very fast search operations. *Journal of Computer and System Sciences*, 28(3):379–394, 1984.

[WM01] Anthony Ian Wirth and Alistair Moffat. Can we do without ranks in burrows wheeler transform compression? In *Data Compression Conference*, pages 419–428, 2001.

[WMB99] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, second edition, 1999.

[WMF94] Marcelo J. Weinberger, Neri Merhav, and Meir Feder. Optimal sequential probability assignment for individual sequences. *IEEE Transactions on Information Theory*, 40:384–396, 1994.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

Biography

Personal

Born in Kitchener-Waterloo, Ontario, Canada, 18 July 1978.

Colleges and Universities

Duke University Durham, NC
Ph.D. in Computer Science, August 2007.

University of Texas at Dallas Richardson, TX
M.S. in Computer Science, May 2000.
B.S. in Computer Science, Summa Cum Laude, May 2000.
B.S. in Mathematics, Summa Cum Laude, May 2000.

Honors and Awards

National Science and Engineering Research Council of Canada (NSERC) Scholarship Winner, 2000-2001.

Excellence in Teaching Assistantship in 1998-1999 and 1999-2000.

College Master's Award for Excellence in Computer Science.

Publications

Alexander Golynski, Roberto Grossi, Ankur Gupta, Rajeev Raman, and Srinivasa Rao. **On the Size of Succinct Indices**. To appear in *Proceedings of European Symposium on Algorithms (ESA)*, Eilat, Israel, October, 2007.

Ankur Gupta, Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. **A Framework for Dynamizing Succinct Data Structures**. To appear in *Proceedings of International Colloquium on Automata, Languages, and Programming (ICALP)*, Wroclaw, Poland, July 2007.

Ankur Gupta, Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. **Compressed Data Structures: Dictionaries and Data-Aware Measures**. To appear in *Proceedings of Theoretical Computer Science (TCS)*, January 2007.

Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. **When Indexing Equals Compression: Experiments With Compressing Suffix Arrays and Applications**. To appear in *Proceedings of the ACM Transactions on Algorithms (TALG)*, January 2007.

Ankur Gupta, Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. **Compressed Dictionaries: Space Measures, Data Sets, and Experiments**. In *Proceedings of the Workshop on Experimental and Efficient Algorithms (WEA)*, Menorca, Spain, May 2006.

Ankur Gupta, Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. **Fully Indexable Data-Aware Dictionaries**. In *Proceedings of the IEEE Data Compression Conference (DCC)*, Snowbird, UT, March 2006.

Luca Foschini, Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. **Fast Compression With a Static Model in High-Order Entropy**. In *Proceedings of the IEEE Data Compression Conference (DCC)*, Snowbird, UT, March 2004.

Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. **When Indexing Equals Compression: Experiments With Compressing Suffix Arrays and Applications**. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, New Orleans, LA, January 2004.

Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. **High-Order Entropy-Compressed Text Indexes**. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Baltimore, MD, January 2003.