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

LUCA FOSCHINI

Scuola Superiore Sant'Anna, Pisa, Italy

ROBERTO GROSSI

Università di Pisa, Pisa, Italy

ANKUR GUPTA

Duke University, Durham, North Carolina

AND

JEFFREY SCOTT VITTER

Purdue University, West Lafayette, Indiana

Abstract. We 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

---

This article is an extended version of Grossi et al. [2004] (invited for this special issue) and of Foschini et al. [2004].

Support for L. Foschini was provided in part by Scuola Superiore Sant'Anna.

Support for R. Grossi was provided in part by the Italian MIUR.

Support for A. Gupta was provided in part by the Army Research Office through grant DAAD19-03-1-0321.

Support for J. S. Vitter was provided in part by the Army Research Office through grant DAAD19-03-1-0321, by the National Science Foundation (NSF) through grant CCR-9877133, and by an IBM research award.

Authors' addresses: L. Foschini, Scuola Superiore Sant'Anna, Piazza Martiri della Libertà 33, 56127 Pisa, Italy, e-mail: foschini@sssup.it; R. Grossi, Dipartimento di Informatica, Università di Pisa, Largo Bruno Pontecorvo 3, 56127 Pisa, Italy, e-mail: grossi@di.unipi.it; A. Gupta, Center for Geometric and Biological Computing, Department of Computer Science, Duke University, Durham, NC 27708-0129, e-mail: agupta@cs.duke.edu; J. S. Vitter, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-2066, e-mail: jsv@purdue.edu.

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`.

Categories and Subject Descriptors: E.1 [**Data**]: Data Structures; E.2 [**Data**]: Data Storage Representations; E.4 [**Data**]: Coding and Information Theory—*Data compaction and compression*; E.5 [**Data**]: Files—*Sorting/searching*; H.3 [**Information Storage and Retrieval**]; F.2 [**Analysis of Algorithms and Problem Complexity**]; I.7.3 [**Document and Text Processing**]: Index Generation

General Terms: Algorithms, Design, Experimentation, Theory

Additional Key Words and Phrases: Entropy, text indexing, Burrows–Wheeler Transform, suffix array

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 Gusfield [1997]. 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 $n \log |\Sigma|$ 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 [Manber and Myers 1993; McCreight 1976] allow fast, general searching of patterns in T in $O(m \log |\Sigma|)$ time, but require $4n \log n$ 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 [Gonnet et al. 1992; Manber and Myers 1993] (that also store the length of the longest common prefix) are nearly as good at searching. Their search time is $O(m + \log n)$ time, but they require a copy of the text; the space cost is only $n \log n$ bits (which can be reduced about 40% in some cases).

Compressed suffix arrays [Grossi and Vitter 2005; Rao 2002; Sadakane 2002, 2003] and opportunistic FM-indexes [Ferragina and Manzini 2001, 2005] 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 [Gonnet et al. 1992]), 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 [Brodnik and Munro 1999; Pagh 2001; Raman et al. 2002].

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.

Grossi and Vitter [2005] developed the compressed suffix array using $2n \log |\Sigma|$ bits in the worst case with $o(m)$ searching time. [Sadakane 2002, 2003] 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 [2001, 2005], 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 et al. [2003] 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 Navarro and Mäkinen [2006] 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.

1.1. OUR RESULTS. In this article, 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 [Grossi et al. 2003] and develop a new design that is driven by experimental analysis for enhanced performance. Briefly, we mention the following new contributions.

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) [Bentley et al. 1986], 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. [2005] 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 $O(n + \min(n, nH_h) \times \log|\Sigma|)$ 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 et al. [2003] have shown how to build the compressed suffix array and FM-index in $O(n \log\log|\Sigma|)$ time. One of our main results in this article 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 article, 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.

1.2. OUTLINE OF ARTICLE. The rest of the article 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, we describe our *wavelet tree* structure, which forms the basis for our compression format `wzip`. In Section 4, we describe a practical implementation of compressed suffix arrays [Grossi and Vitter 2005; Grossi et al. 2004], grounded firmly with theoretical analysis. In Section 5, we discuss a space-efficient implementation of suffix trees. We conclude in Section 6.

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 $[0 \cdots n-1]$ in $\lceil \log \binom{n}{t} \rceil \leq n$ bits, plus additional bits for fast access to the entries. The bound comes from the information-theoretic observation that we need $\lceil \log \binom{n}{t} \rceil$ bits to enumerate each of the $\binom{n}{t}$ possible subsets of $[0 \cdots n-1]$. Equivalently, this is the number of bitvectors B of length n (the universe size) with exactly t **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* [Raman et al. 2002], which support the full repertoire of *rank* and *select* for both **0**s and **1**s in $\lceil \log \binom{n}{t} \rceil + o(n)$ 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

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

As shown in Grossi et al. [2003], the empirical entropy H_0 can be approximated by $\frac{1}{n} \log \binom{n}{t}$. 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 Grossi et al. [2003] 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 [Jacobson 1989; Munro and Raman 1999].

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 $\log \binom{n}{t}$ term.

—The $\log \binom{n}{t}$ term is not necessarily the best possible in practice. As with strings, we can achieve "entropy" bounds that are better than $\log \binom{n}{t} \sim nH_0$.

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 $\log \binom{n}{t}$ 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 $O(\log n)$ bits. Similarly, if D stores $n/2$ elements of a contiguous interval of the universe, we can again represent this information with $O(\log n)$ bits. The $\log \binom{n}{t}$ term treats these two cases the same a random set of $t = n/2$ integers stored in D; thus, the worst-case bound is $\log \binom{n}{n/2} \sim n$ 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 $\log \binom{n}{t}$ 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 [Witten et al. 1999]. 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 $1 + \lfloor \log \ell \rfloor$ in unary, followed by the value of $\ell - 2^{\lfloor \log \ell \rfloor}$ encoded in binary, for a total of $1 + 2\lfloor \log \ell \rfloor$ bits. For example, the γ codes for $\ell = 1, 2, 3, 4, 5, \ldots$ are **1**, **01 0**, **01 1**, **001 00**, **001 01**, ..., respectively. The δ code requires asymptotically fewer bits by encoding $1 + \lfloor \log \ell \rfloor$ via the γ code rather than in unary, thus requiring $1 + \lfloor \log \ell \rfloor + 2\lfloor \log \log 2\ell \rfloor$ bits. For example, the δ codes for $\ell = 1, 2, 3, 4, 5, \ldots$ are **1**, **010 0**, **010 1**, **011 00**, **011 01**, ..., respectively. Byte-aligned codes are another simple encoding for positive integers. Let $lb(\ell) = 1 + \lfloor \log \ell \rfloor$, 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 [Witten et al. 1999] 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 $B_D = \mathbf{0}^{g_1}\mathbf{1}\mathbf{0}^{g_2}\mathbf{1} \cdots \mathbf{0}^{g_t}\mathbf{1}$ and each $g_i \geq 0$. We assume that B_D ends with a **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 **0** and **1**. Using gap encoding we cannot require less than

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

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 $\log \binom{n}{t}$ bits.

FACT 1. *For a conceptual bitvector B_D of known length n, such that B_D ends with a* **1**, *its gap encoding G satisfies $E(G) < \log \binom{n}{t} + 1/2 \log(t(n-t)/n) + \log e\,[(1/(12t) + 1/(12(n-t)) - 1/(12n+1)] + \log\sqrt{2\pi}$, where $t \le n/2$ is the number of* **1**s *in B_D.*

PROOF. By convexity, the worst-case optimal cost occurs when the gaps are of equal length, i.e. $g_i + 1 \le n/t$, giving $E(G) = \sum_{i=1}^{t} lb(g_i + 1) \le t\, lb(n/t) \le t + t\log(n/t) \le (n-t)\log(n/(n-t)) + t\log(n/t)$, since $t \le (n-t)\log(n/(n-t))$ when $t \le n/2$. By Stirling's inequality, $\log \binom{n}{t} > t\log(n/t) + (n-t)\log(n/(n-t)) - 1/2\log(t(n-t)/n) - [(1/(12t) + 1/(12(n-t)) - 1/(12n+1)]\log e - \log\sqrt{2\pi}$, 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 $j \le 2t$ and $\sum_i \ell_i = n$) where either $B_D = \mathbf{1}^{\ell_1}\mathbf{0}^{\ell_2}\mathbf{1}^{\ell_3}\cdots$ or $B_D = \mathbf{0}^{\ell_1}\mathbf{1}^{\ell_2}\mathbf{0}^{\ell_3}\cdots$. (We can determine which case by a single additional bit.) Using run-length encoding, we cannot require less than

$$E(L) = \sum_{i=1}^{j} lb(\ell_i) \tag{2}$$

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 $E(L) < \log \binom{n}{t} + t + 1/2\log(t(n-t)/n) + \log e\,[(1/(12t) + 1/(12(n-t)) - 1/(12n+1)] + \log\sqrt{2\pi}$, where $t \le n/2$ 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, that is, 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 $E(L) \le E(G) + t$. 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. □

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 $E(L) \approx \log \binom{n}{t} \approx 2t = n$. Using RLE twice to encode B_D, we obtain $O(\log n)$ required bits, as indicated by Kolmogorov complexity. On the other hand, finding the Kolmogorov complexity of an arbitrary string is undecidable [Li and Vitanyi 1997].

Despite its theoretical misgivings, we give experimental results on random data in Table I showing that $E(L) \le \log \binom{n}{t}$. 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

TABLE I. COMPARISON BETWEEN RLE ENCODING (RLE+γ), GAP ENCODING (GAP+γ), AND RELATED MEASURES ($\log \binom{n}{t}$, $E(L)$, AND $E(G)$)[a]

$\log(gap)$	RLE+γ	Gap+γ	$\log \binom{n}{t}$	$E(L)$	$E(G)$
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

[a]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.

to encode B_D using the gaps in G represented with the γ code. The fourth column reports the value of $\log \binom{n}{t}$, 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 $E(G)$.

$E(L)$ outperforms $\log \binom{n}{t}$ 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.

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, obtained their Burrows–Wheeler transform (`bwt`), performed the wavelet tree construction on the `bwt` according to the text indexing structure of Grossi et al. [2003], and recorded the sets of integers that need to be stored succinctly. On these sets, we ran the experiments summarized in Table II and Table III. 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 II, each encoding scheme is used in conjunction with RLE to provide the results in the table. (We also report Gap+γ for comparison purposes.) Golomb uses the median value as its parameter b. Maniscalco refers to code [Nelson 2003] that is tailored for use with RLE in `bwt`. Bernoulli is the skewed Bernoulli model with the median value as its parameter b. MixBernoulli uses just one bit to encode gaps of length 1, and for other gap lengths, it uses one bit plus the Bernoulli 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

TABLE II. COMPARISON OF VARIOUS CODING METHODS WHEN USED WITH RUN-LENGTH (RLE) AND GAP ENCODING[a]

File	$E(L)$	$E(G)$	RLE+γ	Gap+γ	RLE+δ	Golomb	Maniscalco	Bernoulli	MixBernoulli
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

[a]Unless stated otherwise, the listed coding method is used with RLE. The files indicated are from the Canterbury Corpus. The values in the table are the bits per symbol (bps) required by each method.

TABLE III. COMPARISON OF VARIOUS CODING METHODS WHEN USED WITH RUN-LENGTH (RLE) ENCODING[a]

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

[a]The files indicated are from the Canterbury and Calgary Corpora. The values in the table are the bits per symbol (bps) required by each method.

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 II, namely just $E(L) + \sum_{i=1}^{j} \lfloor \log \log(2\ell_i) \rfloor$ 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 II 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 [Witten et al. 1999] in Table III. 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 2.3; the sixth column refers to the Huffman code in which the cost of encoding 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 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 2.3, we formalize this experimental finding more clearly by curve-fitting the distribution implied by γ onto the distribution of the run-lengths.

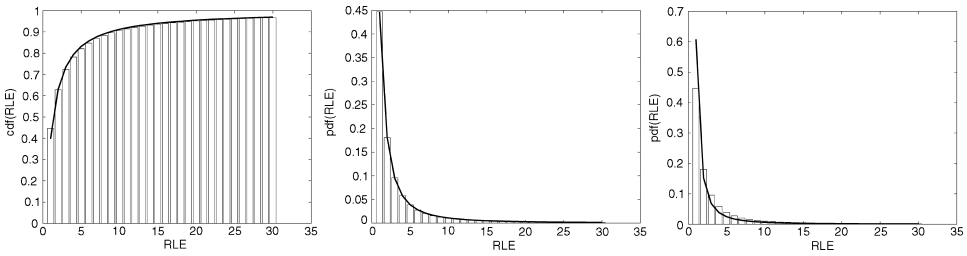

FIG. 1. The x axis shows the distinct RLE values for `bible.txt` in increasing order. Left: The empirical cumulative distribution together with our fitting function *cdf* from (3). Center: The empirical probability density function together with our fitting function *pdf* from (4). Right: The empirical 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.

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 III. 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 [Witten et al. 1999]. 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.

2.3. STATISTICAL EVIDENCE JUSTIFYING THE STATIC MODEL OF 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 1. This distribution is fitted by the function

$$cdf(x) = exp(-a/x) \quad x \in \mathbf{N}^+, \tag{3}$$

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 we obtain the probability density function

$$pdf(x) = \left(\frac{a * \exp(-a/x)}{x^2}\right) \Bigg/ \left(\sum_{i=1}^{\infty} \frac{a * \exp(-a/i)}{i^2}\right), \quad i, x \in \mathbf{N}^+, a \in \mathbf{R}^+ \tag{4}$$

where the term $\sum_{i=1}^{\infty} \frac{a*\exp(-a/x)}{i^2}$ is the normalization factor. As one can see from Figure 1, function (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

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

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.

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), 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 [Schindler 1999]. 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 III 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 [Smith 2003] 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 [2001], 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

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

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

[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.

[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 Moffat et al. [1998].

[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 with respect to a and get

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

where $H(x)$ is the Harmonic mean of the sequence x. By denoting the left hand term by $f(a)$, we have $a = f^{-1}(H(x)^{-1})$. 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 III, 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.

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 $O(\log t)$ 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 $O(\log t)$ 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 $O(\log \log t)$ 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 $select_\mathbf{1}(x)$. 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, 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 $O(p \log t)$ time. This cost is prohibitive: we now describe a method to reduce the time to $O(p + \log t)$ using an idea similar to fractional cascading [Chazelle and Guibas 1986].

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 4.4, the additional space required can be made negligible in practice.

3. *Wavelet Trees*

In this section, we describe the wavelet tree, which forms the basis for both our indexing and compression methods. Grossi et al. [2003] introduce the *wavelet tree* for reducing the redundancy inherent in maintaining separate dictionaries for each symbol appearing in the text. To remove redundancy among dictionaries, 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 time for a particular item is now 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 2, built on the `bwt` of the text `mississippi#`, where `#` is an end-of-text symbol.

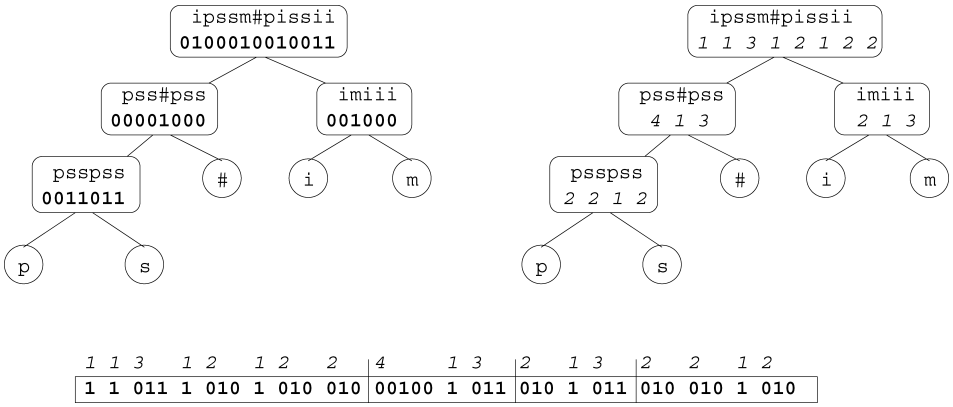

FIG. 2. Left: An example wavelet tree. Right: an RLE encoding of the wavelet tree. Bottom: actual encoding in memory of the right tree in heap layout with γ encoding.

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 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 $O(\log|\Sigma|)$ 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 Grossi et al. [2003] 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 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 Grossi et al. [2003]. (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.

TABLE IV. EFFECT ON PERFORMANCE OF WAVELET TREE USING FRACTIONAL CASCADING AND/OR A HUFFMAN PREFIX TREE SHAPE[a]

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

[a]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.)

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 $1 + \log \sum_x f(x)/f(c)$, which is nearly the optimal average access cost to c. Thus, on average, we require $1 + \log \sum_x f(x)/f(c)$ calls to *rank* or *select* involving leaf c.

LEMMA 1. *Given a distribution of accesses to the wavelet tree in terms of the estimated 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 $1 + \log \sum_x f(x)/f(c)$. The worst-case space occupancy of the wavelet tree does not change.*

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 1), reducing the weighted average depth of the wavelet tree to $H_0 \leq \log |\Sigma|$. We performed experiments to demonstrate the effectiveness of fractional cascading and the Huffman-style tree shaping. Some results are summarized in Table IV. 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 V for some text files from the Canterbury and Calgary Corpora [2001], and some new files available on TREC Tipster 3 [2000]. Our results for `wave` are in the second column. The arithmetic

TABLE V. WAVELET TREE WITH RLE+γ ENCODING AS A PLAIN 0-ORDER COMPRESSOR (COLUMN `wave`) AND APPLIED TO THE `bwt` STREAM (COLUMN `wzip`)[a]

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**

[a]Remaining columns are for other compressors. The values in the table are in bits per symbol (bps).

code [Rissanen and Langdon 1979] 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 [Oki 2003]; 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 V 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 V.

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`.

3.1. 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 $O(n + \min(n, nH_h) \times \log|\Sigma|)$ time. Directories that enable fast access to our wavelet tree can be created in the same time. 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 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 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.

3.2. COMPRESSION WITH `bwt2wzip`. In this section, we describe our compression method `bwt2wzip`, which takes as input the `bwt` stream (the Φ function in Grossi et al. [2003]) 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 $O(n + \min(n, nH_h) \times \log|\Sigma|)$ 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 $O(\log|\Sigma|)$ time, it requires $O(n \log|\Sigma|)$ time in total. We describe a refinement of this construction method requiring $O(n + \min(n, nH_h) \times \log|\Sigma|)$ 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 $O(n_r \times \log|\Sigma|)$, to perform the n_r traversals of the wavelet tree, taking $O(\log|\Sigma|)$ 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 2 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 Grossi et al. [2003], and is proved in Section 4.2.

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.

3.4. 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 VI 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 VI 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).

TABLE VI. RUNNING TIMES FOR `bwt2wzip` AND `wzip2bwt` NORMALIZED WITH THAT OF A SIMPLE COPY ROUTINE[a]

	`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

[a]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`.

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 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 4.

It is worth noting that the total cost of compression and decompression is much larger than what discussed so far, once we take into account the cost of suffix sorting in order 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`).

4. *Exploiting Suffix Arrays: Indexing Equals Compression*

We explored dictionary methods which perform well in practice. Now, we apply these dictionary methods to compressed suffix arrays [Grossi et al. 2003; Grossi and Vitter 2005; Sadakane 2002, 2003] and show both experimental success as well as a theoretical analysis of these practical methods. First, we provide some background notions from Grossi and Vitter [2005] and Grossi et al. [2003].

4.1. COMPRESSED SUFFIX ARRAYS (CSA).

To recap, a standard suffix array [Gonnet et al. 1992; Manber and Myers 1993] 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, $T\left[SA[i], n\right]$. The size of a suffix array is $\Theta(n \log n)$ bits, as each of the positions stored uses $\log n$ bits. A suffix array allows constant time *lookup* to $SA[i]$ for any i. The compressed suffix array [Grossi and Vitter 2005] contains the same information as a standard suffix array.

Definition 1. Given a text T of length n, a *compressed suffix array* [Grossi and Vitter 2005; Sadakane 2002, 2003] 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 $1 \leq i \leq n$; *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 $1 \leq i \leq n$ and $1 \leq c \leq n - SA[i] + 1$.

The data structure is recursive in nature, where each of the $\ell = \log\log n$ 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 $0 \leq k < \log\log n$, 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, $\ell = \log\log n$ 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 $SA_k[i] = n$):

$$\Phi_k(i) = \{\ j \quad \text{such that } SA_k[j] = (SA_k[i] \mod n) + 1\ \}. \tag{5}$$

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

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

The representation of B_k and $rank(B_k, i)$ 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 [Ferragina et al. 2005] for contexts of length at most h. These sublists $\langle x, y \rangle$ can be stored by succinct dictionaries using $\log \binom{n_k^x}{n_k^{x,y}}$ 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 Grossi et al. [2003] for further details on the consequences of this observation with regard to compression.

4.2. PRACTICAL CONSIDERATIONS FOR COMPRESSED SUFFIX ARRAYS. In this section, we apply our practical dictionaries to the CSA framework we described

in Section 4.1, achieving practical data structures that implicitly achieve at most twice the high-order entropy of the text.

THEOREM 1. *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 $E(L) + \sum \log(g_i + 1)$ bits of space (since they are RLE+gamma dictionaries). Since $E(L) \leq E(G) + t$ by Fact 1 and $E(G) = \sum \log(g_i + 1) + t$ 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 Grossi et al. [2003], at most doubling the theoretical worst-case bounds. The result follows automatically from the analysis in Grossi et al. [2003]. □

This discovery brings up a remarkable point—our practical dictionary is blind to the universe size that was so carefully constructed in Grossi et al. [2003] to allow the use of the fully indexable dictionaries from Raman et al. [2002] (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^* [Ferragina et al. 2005]) for $h \geq 0$, where $|P_h| \leq n^\alpha$, for some $0 < \alpha < 1$. (This reasonable assumption is also used in [Grossi et al. 2003].) We argue that due to the nature of our directory, we are still able to achieve the higher-order entropy given in Grossi et al. [2003]. Said more mathematically, we can split the cost in Grossi et al. [2003] 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 Eq. (5).

THEOREM 2. *We can encode 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 Grossi et al. [2003], 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 that is, $x_j = x = x_{j+1}$. 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 Grossi et al. [2003] when the sublists are treated separately (since they never encode a gap that traverses a sublist). Since $\log g_j \leq \log(g'_j + g''_j) + \log g'''_j$, we can bound our total overhead by

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

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 $\log(g'_j + g''_j) = O(\log n)$ bits, we can bound this cost by the number of sublists among the entire structure of Grossi et al. [2003]. 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 Grossi et al. [2003]. 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 $\log \binom{n}{c} = o(n)$ 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 1, thus doubling the entropy term and proving the claimed bound. □

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 [Bentley et al. 1986] (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

TABLE VII. MEASURE OF THE EFFECT OF MTF ON VARIOUS CODING METHODS WHEN USED WITH RLE[a]

File	MTF	$E(L)$	γ	δ	Golomb	Maniscalco	Bernoulli	MixBernoulli
`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

[a]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.

`bzip2`, without using MTF, arithmetic, or multi-table Huffman encoding. (See also Wirth and Moffat [2001].) 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 VII, 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 [2000]. Each row represents duplicated experiments performed as follows. (Figure 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, Golomb code, Maniscalco code, Bernoulli code, or MixBernoulli 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 (2) in Section 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 VII report the resulting bps figures for the γ, δ, Golomb, Maniscalco, Bernoulli, and MixBernoulli codes. Golomb uses the median value as its parameter b; Maniscalco refers to code [Nelson 2003]; Bernoulli is the skewed Bernoulli model with the median value as its parameter b; MixBernoulli uses just one bit to encode gaps of length 1, and for other gap lengths, it uses one bit plus the Bernoulli code.

Table VII 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

TABLE VIII. COMPARISON OF SPACE REQUIRED BY Φ AND THE COMPRESSED SUFFIX ARRAY (CSA)[a]

	book1	bible.txt	E.coli	world192.txt	ap90-64.txt	ap90-100.txt
Φ overhead	0.166/0.171	0.050/0.052	0.050/0.051	0.067/0.069	0.032/0.033	0.032/0.033
Φ	2.785/2.790	1.681/1.683	2.231/2.232	1.586/1.588	1.700/1.701	1.659/1.660
CSA overhead	0.328/0.332	0.210/0.212	0.210/0.212	0.228/0.230	0.192/0.194	0.191/0.192
CSA	**2.946/2.951**	**1.841/1.843**	**2.391/2.392**	**1.747/1.749**	**1.860/1.861**	**1.818/1.819**

[a]Overhead refers to all space other than the RLE+γ encoding for the data itself. The values in the table are given in bits per symbol (bps). Entries contain two values—the first is tuned for space, the second is tuned for speed.

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 $1 + \lfloor (x-1)/b \rfloor + \lfloor \log b \rfloor$ 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.

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 VIII, we show the number of bits per symbol needed for compressed suffix arrays on some files from the Canterbury corpus and TREC Tipster 3 [2000]. We incur a minimal overhead cost for adding suffix array functionality. Note the small difference between the split entries in our method; the additional space implements fractional cascading in our wavelet tree, and requires almost negligible space.

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 [Kurtz 1999]. Suffix trees are at the heart of many algorithms on strings and sequences, so their full functionality is needed [Gusfield 1997]. 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 $O(n \log |\Sigma|)$ bits or $|CSA| + 6n + o(n)$ bits (Kunihiko Sadakane 2002, personal communication), 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 [Sadakane 2002]. 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 [Munro et al. 2001], though there is some slowdown since *LCP* information is not stored explicitly.[5] In either case, a separate

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

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

(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 $S = O(\log n)$. We implement part of the lowest common ancestor simplification introduced in Bender and Farach-Colton [2004]. 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 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, for example, 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 [Abouelhoda et al. 2004; Arimura et al. 2001]. For each node u, there are two integers $1 \leq u_l \leq u_r \leq n$ such that $SA[u_l \cdots u_r]$ 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 [Gusfield 1997].

We use Kasai et al.'s [2001] linear-time method to compute *LCP* information. We modify Sadakane's method [Sadakane 2002] to store only *LCP* values larger than $2 \log n$; it works and compresses well. We also implement the doubling technique of Bender and Farach-Colton [2004] to compute *LCP* information in constant time, though we can trade time to reduce the space required.

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

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 determine 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 3.4. The data sets used were drawn mainly from the Canterbury corpus, the TREC Tipster 3 [2000], and 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 2). Module `phi` implements the wavelet tree and its use in compressed suffix arrays (Section 3), while module `csa` implements the compressed suffix array and related functionality (Section 4). Module `lcp` stores LCP information and module `st` implements suffix tree functionality, though we avoid storing any nodes explicitly (Section 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.

6. *Conclusions*

In this article, 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 [Hon et al. 2004]. 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, for example, 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 Deorowicz [2002], though the analysis in Grossi et al. [2003] 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 Deorowicz [2002], Fenwick [1996, 2002] and Howard [1997]) present other refinements with various tradeoffs. Theoretical exploration of the suite of algorithms from Deorowicz [2002] 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 (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.

ACKNOWLEDGMENTS. The authors would like to thank Raffaele Giancarlo, Giovanni Manzini, and Rajeev Raman for helpful comments. We would also like to thank Kunihiko Sadakane for fruitful discussions at the 2002 DIMACS Workshop "Data Compression in Networks and Applications".

REFERENCES

ABOUELHODA, M. I., KURTZ, S., AND OHLEBUSCH, E. 2004. Replacing suffix trees with enhanced suffix arrays. *J. Disc. Algor. 2*, 1, 53–86.

ARIMURA, H., ASAKA, H., SAKAMOTO, H., AND ARIKAWA, S. 2001. Efficient discovery of proximity patterns with suffix arrays (extended abstract). In *CPM: 12th Symposium on Combinatorial Pattern Matching*.

BENDER, M. A., AND FARACH-COLTON, M. 2004. The level ancestor problem simplified. *Theoret. Comput. Sci. 321*, 1, 5–12.

BENTLEY, J., SLEATOR, D., TARJAN, R., AND WEI, V. 1986. A locally adaptive data compression scheme. *Commun. ACM*, 320–330.

BRODNIK, A., AND MUNRO, J. I. 1999. Membership in constant time and almost-minimum space. *SIAM J. Comput. 28*, 5 (Oct.), 1627–1640.

THE CANTERBURY CORPUS. 2001. http://corpus.canterbury.ac.nz.

CHAZELLE, B., AND GUIBAS, L. J. 1986. Fractional cascading: I. A data structuring technique. *Algorithmica 1*, 2, 133–162.

DEOROWICZ, S. 2002. Second step algorithms in the Burrows-Wheeler compression algorithm. *Softw. Pract. Exper. 32*, 99–111.

FENWICK, P. 1996. Punctured elias codes for variable-length coding of the integers. The University of Auckland, NZ. TR 137. ISSN 1173–3500.

FENWICK, P. 2002. Burrows-wheeler compression with variable-length integer codes. *Softw. Pract. Exper. 32*, 1307–1316.

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

FERRAGINA, P., AND MANZINI, G. 2001. An experimental study of an opportunistic index. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM, New York, pp. 269–278.

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

FOSCHINI, L., GROSSI, R., GUPTA, A., AND VITTER, J. S. 2004. Fast compression with a static model in high-order entropy. In *Proceedings of the IEEE Data Compression Conference* (Snowbird, UT, Mar.)

GONNET, G. H., BAEZA-YATES, R. A., AND SNIDER, T. 1992. New indices for text: PAT trees and PAT arrays. In *Information Retrieval: Data Structures and Algorithms*. chap. 5. Prentice-Hall, Englewood Cliffs, NJ, pp. 66–82.

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

GROSSI, R., GUPTA, A., AND VITTER, J. S. 2004. When indexing equals compression: Experiments with compressing suffix arrays and applications. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms*, ACM, New York.

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

GUSFIELD, D. 1997. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, Cambridge, MA.

HON, W., LAM, T., TSE, W., WONG, C., AND YIU, S. 2004. Practical aspects of compressed suffix arrays and fm-index in searching dna sequences. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX)*.

HON, W.-K., SADAKANE, K., AND SUNG, W.-K. 2003. Breaking a time-and-space barrier in constructing full-text indices. In *Proceedings of the 44th Annual IEEE Symposium on Foundation of Computer Science*. IEEE Computer Society Press, Los Alamitos, CA pp. 251–260.

HOWARD, P. G. 1997. Interleaving entropy codes. In *Sequences*.

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

KASAI, T., LEE, G., ARIMURA, H., ARIKAWA, S., AND PARK, K. 2001. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Combinatorial Pattern Matching (CPM)*. 181–192.

KURTZ, S. 1999. Reducing the space requirement of suffix trees. *Softw. Pract. Experi. 29*, 13, 1149–1171.

LI, M., AND VITANYI, P. 1997. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag, New York.

MANBER, U., AND MYERS, G. 1993. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput. 22*, 5, 935–948.

MCCREIGHT, E. M. 1976. A space-economical suffix tree construction algorithm. *J. ACM 23*, 2, 262–272.

MOFFAT, A., NEAL, R. M., AND WITTEN, I. H. 1998. Arithmetic coding revisited. *ACM Trans. Inf. Syst. (TOIS) 16*, 3, 256–294.

MUNRO, J. I., AND RAMAN, V. 1999. Succinct representation of balanced parentheses, static trees, and planar graphs. *SIAM J. Comput. 31*, 762–776.

MUNRO, J. I., RAMAN, V., AND SRINIVASA RAO, S. S. 2001. Space efficient suffix trees. *J. Algorithms 39*, 205–222.

NAVARRO, G., AND MÄKINEN, V. 2006. Compressed full-text indexes. Tech. Rep. TR/DCC-2006-6, University of Chile.

NELSON, M. 2003. Run length encoding/RLE. http://www.datacompression.info/RLE.shtml.

OKI, M. 2003. http://www.infor.kanazawa-it.ac.jp/∼ishii/lhaunix/.

PAGH, R. 2001. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput. 31*, 353–363.

RAMAN, R., RAMAN, V., AND RAO, S. S. 2002. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*. 233–242.

RAO, S. S. 2002. Time-space trade-offs for compressed suffix arrays. *IPL 82*, 6, 307–311.

RISSANEN, J., AND LANGDON, G. G. 1979. Arithmetic coding. *IBM J. Res. Devel. 23*, 2 (Mar.), 149–162.

SADAKANE, K. 2002. Succinct representations of *lcp* information and improvements in the compressed suffix arrays. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM, New York.

SADAKANE, K. 2003. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms 48*, 2, 294–313.

SCHINDLER, M. 1999. http://www.compressconsult.com/rangecoder.
SMITH, J. O., III. 2003. http://ccrma-www.stanford.edu/~jos/mdft/Autocorrelation. html.
TREC. TIPSTER 3. 2000. http://trec.nist.gov/data/docs_eng.html.
WIRTH, A. I., AND MOFFAT, A. 2001. Can we do without ranks in burrows wheeler transform compression? In *Data Compression Conference*. pp. 419–428.
WITTEN, I. H., MOFFAT, A., AND BELL, T. C. 1999. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan-Kaufmann, Los Altos, CA.

RECEIVED JUNE 2004; REVISED JUNE 2006; ACCEPTED JUNE 2006