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

Roberto Grossi* Ankur Gupta† Jeffrey Scott Vitter‡

Abstract

We report on a new and improved version of high-order entropy-compressed suffix arrays, which has theoretical performance guarantees comparable to previous work, yet 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—and support fast and powerful searches. To our knowledge, this is the best known method in terms of space for fast searching. We can additionally use a simple notion to encode and decode block-sorting transforms (such as the Burrows-Wheeler transform), achieving a slightly better compression ratio than `bzip2`. We also provide a compressed representation of suffix trees (and their associated text) in a total space that is comparable to that of the text alone compressed with `gzip`.

1 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 [Gus97]. (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.) Inverted files do not offer as much functionality, but they provide excellent index compression, requiring about $0.15n \log |\Sigma|$ bits of space in practice [MZ96, WMB99]. However, inverted files also require a separate copy of the text. In terms of functionality, inverted files support efficient search only for words (or parts of words) in the text; they cannot search efficiently for arbitrary substrings of T, as required in biological sequences, documents written in Eastern languages, or phrase searching [Goo]. An efficient combination of inverted file compression, block addressing, and sequential search on word-based Huffman compressed text is described in [NSN+00].

The suffix tree is a much more powerful text index (in the form of a compact trie) whose leaves store each of the n suffixes contained in the text T. Suffix trees [MM93, McC76] allow fast, general search 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 [GBS92, MM93] (also storing 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 in some cases can be reduced about 40%).

*Dipartimento di Informatica, Università di Pisa, Largo Pontecorvo 1, 56127 Pisa (`grossi@di.unipi.it`). Support was provided in part by the Italian MIUR project "ALINWEB: Algorithmics for Internet and the Web" and by the French EPST program "Algorithms for Modeling and Inference Problems in Molecular Biology".

†Center for Geometric and Biological Computing, Department of Computer Science, Duke University, Durham, NC 27708–0129 (`agupta@cs.duke.edu`). Support was provided in part by the Army Research Office through grant DAAD19–03–1–0321.

‡Department of Computer Sciences, Purdue University, West Lafayette, IN 47907–2066 (`jsv@purdue.edu`). Support was provided in part by the Army Research Office through grant DAAD19–03–1–0321, by the National Science Foundation through grant CCR–9877133, and by an IBM research award.

Compressed suffix arrays [GV00, Rao02, Sad00, Sad02b] and opportunistic FM-indexes [FM00, FM01] represent modern trends in the design of advanced indexes for full-text searching of documents, in that they support the functionalities of suffix arrays and suffix trees, which are more powerful than classical inverted files [GBS92], yet they also 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.

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, we support more functionality than standard text compression.

Grossi and Vitter [GV00] developed the compressed suffix array using $2n \log |\Sigma|$ bits in the worst case with $o(m)$ searching time. Sadakane [Sad00, Sad02b] extended its functionalities to be self-indexing, and related the space bound to the order-0 empirical entropy H_0. Ferragina and Manzini devised the FM-index [FM00, FM01], which is based on the Burrows-Wheeler transform (`bwt`) and is the first to encode the index size with respect to the hth-order empirical entropy H_h of the text, encoding in $(5 + \epsilon)nH_h + o(n)$ bits. Navarro [Nav, Nav04] recently developed an index requiring $4nH_h + o(n)$ bits. Grossi, Gupta, and Vitter [GGV03] exploited the higher-order entropy H_h of the text to represent a compressed suffix array in just $nH_h + o(n)$ bits. The index is optimal in space, apart from lower-order terms, achieving asymptotically the empirical entropy of the text (with a multiplicative constant 1).

The above self-indexes are so powerful that the text is implicitly encoded in them and is not needed explicitly. Searching needs to decompress a negligible portion of the text and is competitive with previous solutions. In practical implementation, these new indexes occupy around 25–40% of the text size and do *not* need to keep the text itself. If the text is highly compressible so that $H_h = o(1)$ and the alphabet size is small, the text index in [GGV03] provides $o(m)$ search time using only $o(n)$ bits; several theoretical tradeoffs between space and search time were also developed.

1.1 Our Results

In this paper, we provide an experimental study of compressed suffix arrays in order to evaluate their practical impact. (This paper is an extended version of [GGV04].) In doing so, we exploit the properties and intuition of our earlier result [GGV03] and develop a new theoretical design with enhanced practical performance. Briefly, we mention the following new contributions.

We provide a new practical implementation of succinct dictionaries that takes less space than the worst case. We then use these dictionaries (organized in a *wavelet tree*) to achieve a simplified "encoding" for high-order contexts, along with run-length encoding (RLE) and γ encoding. This construction shows that Move-to-Front (MTF) [BSTW86], arithmetic, and Huffman encoding are not strictly necessary to achieve high-order compression with the Burrows-Wheeler Transform (`bwt`). While recent work of Giancarlo and Sciortino [GS03] and Ferragina and Manzini [FM04] show how to find an optimal partition of the `bwt` to attain this goal, our encoding leads implicitly to an optimal partition.

We then extend the wavelet tree so that its search can be sped up by fractional cascading and exploiting a-priori distributions 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, uniform data should not only be more compact when compressed, but should also require less time to index and compress.) Recently Hon, Sadakane, and Sung have shown how to build the compressed suffix array and FM-index in $O(n \log \log |\Sigma|)$ time [HSS03], showcasing yet another tradeoff from our method. We couple our tasks with an elegant analysis of high-order entropy using our simple encoding, as well as highlighting the importance of the underlying statistical model.

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 slightly better than `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 paper, 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 term of a percentage, it suffices to multiply bps by 100/8.

1.2 Outline of Paper

The rest of the paper is organized as follows. In the next section, we built 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 [GV00, GGV03], grounded firmly in 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

Succinct dictionaries [BM99, Pag01] are constant-time rank and select data structures occupying tiny space. They store t entries chosen from a bounded universe $[0 \dots n-1]$ (or any translation of it) in $\lceil \log \binom{n}{t} \rceil \leq n$ bits, plus additional bits for their internal directories. The bound comes from the information theoretic observation that we need this number of bits to enumerate each of the possible $\binom{n}{t}$ subsets of $[0 \dots n-1]$. Equivalently, this is the number of bitvectors B of length n (the universe size) with exactly t **1**s, so 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 ith bit in B can be computed as $B[i] = rank_{\mathbf{1}}(i) - rank_{\mathbf{1}}(i-1)$. The constant-time fully indexable dictionaries [RRR02] 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 probability of finding a **1** in bitvector B, and $p(\mathbf{0}) = 1 - p(\mathbf{1})$. We define the empirical entropy H_0 as

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

When t is small with respect to n, the empirical entropy H_0 can be approximated by $\frac{1}{n} \log \binom{n}{t}$. Thus, we can think of succinct dictionaries (such as [RRR02]) as 0th-order compressors that can also retrieve any individual bit in constant time. Hence succinct dictionaries are not only of theoretical interest, but they provide the basis for space-efficient representation of trees and graphs [Jac89, MR97]. Recently, dictionaries have been shown to be crucial for text indexing data structures. Specifically, the data structuring framework in [GGV03] uses suffix arrays to transform dictionaries into high-order entropy compressing text indexers. As a result, we stress the important consideration of dictionaries in practice, since they can contribute fast access to data as well as solid, effective, and alternative compression. In particular, such dictionaries avoid a complete sequential scan of the data when retrieving portions of it.

2.1 Practical Dictionaries

In this section, we explore practical alternatives to dictionaries for 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 related to improving the access time to data, is non-negligible and can completely dominate the $\log \binom{n}{t}$ term.

- The $\log \binom{n}{t}$ term is not necessarily the best possible in practice. As in strings, we can achieve high-order empirical entropy bounds, which 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 still represent this information with $O(\log n)$ bits. The $\log \binom{n}{t}$ bound accounts for these two cases in the same way as what happens for a random set of $t = n/2$ integers stored in D, thus giving $\log \binom{n}{n/2} \sim n$ 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 in the examples above allows significant compression. 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 [WMB99]. We briefly describe those relevant notions.

Run-length encoding (RLE) simply 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, there is no need to encode s, since its value will alternate between **0** and **1**.

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 $\mathbf{1}, \mathbf{01\,0}, \mathbf{01\,1}, \mathbf{001\,00}, \mathbf{001\,01}, \ldots$, respectively. The δ code requires fewer bits asymptotically 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 $\mathbf{1}, \mathbf{010\,0}, \mathbf{010\,1}, \mathbf{011\,00}, \mathbf{011\,01}, \ldots$, respectively. Byte-aligned codes are another simple encoding for positive integers that are not too small. Let $lb(\ell) = 1 + \lfloor \log \ell \rfloor$, the minimal number of bits required to represent the positive integer ℓ. Byte-aligned code splits the $lb(\ell)$ bits into groups of 7 bits each, prepending a "continuation" bit as most significant to indicate whether there are more bits of ℓ in the next byte. We refer to [WMB99] for other encodings.

We can represent a conceptual bitvector B_D by a vector of nonnegative "gaps" $G = \{g_1, g_2, \ldots, g_t\}$, where $B_D = \mathbf{0}^{g_1}\mathbf{1}\mathbf{0}^{g_2}\mathbf{1}\ldots\mathbf{0}^{g_t}\mathbf{1}$, where 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**.

Let the "optimal cost" of the dictionary using gap encoding be denoted by $E(G) = \sum_{i=1}^{t} lb(g_i + 1)$. The important point is that the optimal cost is never worse than the optimal worst-case encoding of B_D, which takes $\log \binom{n}{t}$ bits.

Fact 1 *The optimal cost of a dictionary using gap encoding satisfies $E(G) \leq \log \binom{n}{t}$, where $t \leq n/2$ is the number of* **1***s out of a universe of size n.*

Proof: Let's assume for the moment that B_D ends with a **1**. By convexity, the worst-case optimal cost occurs when the gaps are of equal length, giving $E(G) \leq \sum_{i=1}^{t} lb(g_i + 1) \leq t\, lb(n/t) \leq t + t\log(n/t) \leq \log \binom{n}{t}$, which follows since $\log \binom{n}{t} = t\log(n/t) + t\log e - \Theta(t/n) + O(\log t)$. If B_D doesn't end with a **1**, we need to append one extra bit and $lb(g_{t+1} + 1)$ bits, and the bound still holds by a similar argument. □

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 run-length values $L = \{\ell_1, \ell_2, \ldots, \ell_j\}$ (with $j \leq 2t$ and $\sum_i \ell_i = n$) where either $B_D = \mathbf{1}^{\ell_1}\mathbf{0}^{\ell_2}\mathbf{1}^{\ell_3}\ldots$ or $B_D = \mathbf{0}^{\ell_1}\mathbf{1}^{\ell_2}\mathbf{0}^{\ell_3}\ldots$. (We can determine which case by a single additional bit.) We can define the optimal cost of the dictionary using run-length encoding as $E(L) = \sum_{i=1}^{j} lb(\ell_i)$. By a similar argument, we can prove the following:

$\log(gap)$	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

Table 1: Comparison between $E(L)$, RLE+γ codes, and $\log \binom{n}{t}$. 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.

Fact 2 *The optimal cost of a dictionary using run-length encoding satisfies* $E(L) \leq \log \binom{n}{t} + (2 - \log e)t$, *where* $t \leq n/2$ *is the number of* **1**s *out of a universe of size* n.

Proof: It follows from the fact $E(L) \leq E(G) + t$. □

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 $E(L) \sim \log \binom{n}{t} \sim 2t = n$. 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 [LV97]. Despite its theoretical misgivings, we give experimental results on random data in Table 1 showing $E(L) \leq \log \binom{n}{t}$. Notice that $E(L)$ outperforms $\log \binom{n}{t}$ for real data sets, indicating that there must be few enough situations with a singleton **1** (or **0**) that their cost is more than paid for by the improved coding of contiguous items. Notice also that RLE+γ outperforms Gap+γ for small gap sizes (namely 4 or less). The columns for RLE+γ and Gap+γ codes refer to taking B_D and encoding each run-length (or gap) using the γ code, rather than *lb* (which is not a prefix code). This behavior motivates our choice for RLE, as many gap sizes are small in our distributions.

2.2 Empirical Distribution of RLE Values and γ Codes

We performed experiments comparing the space occupancy of several different encodings in place of the γ code. We summarize those experiments in Table 2. Each of the encoding schemes is used in conjunction with RLE (unless noted otherwise) to provide the results in the table. Golomb uses the median value as its parameter b. Maniscalco refers to code [Nel] that is specially tailored for RLE in the Burrows-Wheeler transform (`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. Note also that $E(G)$ and Gap+γ outperform their respective counterparts on `random.txt`, which represents the worst case for RLE.

We can use δ codes to store B_D as in Table 2, using just $E(L) + \sum_{i=1}^{j} \lfloor \log \log(2\ell_i) \rfloor$ bits by Fact 2. Similarly, γ coding requires $2E(L) - t$ bits, though in practice it outperforms δ, since γ is more efficient for small run-lengths. For a detailed study on the encoding of arbitrary integers sequences, we refer the reader to [WZ99]. In this paper, we focus on encoding RLE values for the dictionary problem. Table 2 suggests γ as best encoding to couple with RLE.

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

Table 2: Comparison of various coding methods when used with run-length (RLE) and gap encoding on the `bwt` stream for each file listed. Unless stated otherwise, the listed coding method is used with RLE. The files indicated are from the Canterbury Corpus [Can]. The values in the table are the bits per symbol (bps) required by each method.

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

Table 3: Comparison of various coding methods when used with run-length (RLE) encoding on the wavelet tree stream. The files indicated are from the Canterbury and Calgary Corpuses [Can]. The values in the table are the bits per symbol (bps) required by each method.

A natural question arises as to the choice of the simplistic γ encoding, since theoretically speaking, a number of other prefix codes (δ, ζ, and skewed Golomb, for instance) outperform γ codes. However, γ encoding seems extremely robust according to the experiments above. We consider further comparisons with fractional coding and Huffman prefix codes [WMB99] in Table 3 on the wavelet tree stream (see Section 3). In the table, the 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 mentioned 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 again reinforce the observation that γ encoding is nearly the best. (In Section 2.3, we will formalize this experimental finding more clearly by curve-fitting the distribution implied by γ onto the distribution of the run-lengths.)

Improving upon γ to encode these RLE values requires a significant amount of work with more complicated methods. For the purposes of illustration, consider the comparison of γ encoding to that of an optimal Huffman encoding, given in Table 3. Note that the γ code differs from Huffman 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. As a matter of fact, this news is both encouraging and discouraging. It seems that there is no real hope to improve upon γ using prefix codes, since Huffman codes are optimal prefix codes [WMB99]. Further improvement then, in some sense, necessitates more complicated

techniques (such as arithmetic coding) which have their own host of difficulties, most often a greatly increased encoding/decoding time.

2.3 Statistical Evidence Justifying the Static Model of γ Codes

In this section,[1] we motivate our choice of γ encoding more formally, with statistical evidence suggesting that the underlying distribution of RLE values matches that which is optimally encoded by γ encoding (or equivalently Bernoulli, with $b = 1$). For instance, consider the empirical cumulative distribution of the RLE values for `bible.txt`, shown in Figure 1. Note that this distribution is fitted by the function

$$cdf(x) = e^{-a/x} \qquad x \in \mathbf{N}^+, \tag{1}$$

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

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

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

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

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

Since the γ code is optimal for distributions proportional to $1/x^2$, we finally have some reasonable motivation for the success of the γ code on an RLE stream of wavelet tree data. 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[3], but the results were not satisfying when compared with γ. To resolve this problem, we employ the statistical model of function *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 (1), 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 called range coder [Sch], employed in `szip`. We encode the RLE length ℓ by assigning it an interval of length $cdf(\ell+1) - cdf(\ell) = pdf(\ell)$. (This encoding appears to be faster than using the cumulative counts of the frequency of values already scanned, like other well-known arithmetic coders.) With this kind of compressor, we improve the compression rate from 1–5% with respect to γ encoding (see Table 3). We then transform our arithmetic compressor so that the parameter a could be changed

[1] Section 2.3 is part of Luca Foschini's thesis, "Studio di un metodd efficiente nella compresione dei dati", University of Pisa, 2004.

[2] 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.

[3] 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 [MNW95].

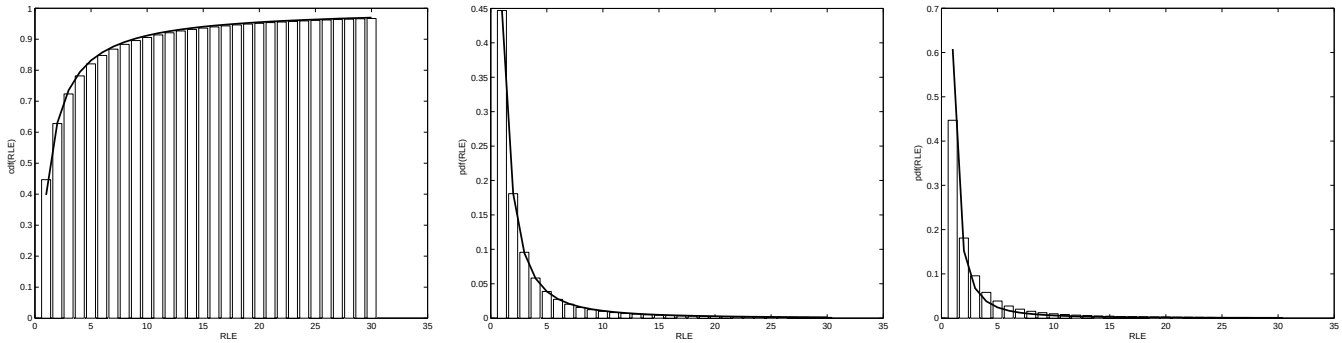

Figure 1: The x axis shows the distinct RLE values for `bible.txt`, in increasing order. Left: The empirical cumulative distribution together with our fitting function *cdf* from (1). Center: The empirical probability density function together with our fitting function *pdf* from (2). Right: The empirical 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.

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.) Though we could not find a satisfiable measure for this notion, we compute the (normalized) autocovariance of the RLE values. This method is widely adopted in signal theory [AUT] as a good indicator of independence of a sequence of values, though it does not necessarily imply independence. In our case, the correlation between consecutive RLE values is very low for the files in Canterbury corpus, which again, though it does not imply independence in the strict sense, is a strong indication nonetheless. With this observation in mind, we assume statistical independence of the RLE values in order to define the likelihood function

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

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 L_x with respect to a and get

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

where $H(x)$ is the Harmonic mean of the sequence x. By denoting the left hand term by $f(a)$, we have $a = f^{-1}\big(H(x)^{-1}\big)$. Unfortunately, $f(\cdot)$ is not an analytical function and is very difficult to compute, even for fixed a. For instance, for $a = 0$ 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 (in which $a = 0.88$). Looking again at Table 3, the improvement is negligible, ranging from 1–2% at best. The best case is the file `random.txt` (in the Calgary corpus), for which the hypothesis of independence of RLE values holds with high probability by its very construction.

2.4 Fast Access of Practical Dictionaries

In order to support fast access to our RLE+γ encoding, we can use a simple scheme from [BM99, Jac89, Pag01, RRR02] and pay an additional cost of $o(n)$ bits. As previously noted, however,

such schemes have limited use in practice because the $o(n)$ space required to support fast access is non-negligible and often contributes the bulk of space. In addition, that space bound does not scale well in our text indexing data structures.

In this section, we focus on the practical implementation of our scheme to avoid the cost of the directories by relaxing the access cost. We propose a simplified version of the data structure that exploits the specific distribution of the 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. We represent B_D by the vector of run-length values $L = \{\ell_1, \ell_2, \ldots, \ell_j\}$ (with $j \leq 2t$ and $\sum_i \ell_i = n$) where either $B_D = \mathbf{1}^{\ell_1}\mathbf{0}^{\ell_2}\mathbf{1}^{\ell_3}\ldots$ or $B_D = \mathbf{0}^{\ell_1}\mathbf{1}^{\ell_2}\mathbf{0}^{\ell_3}\ldots$. (We can determine which case by a single additional bit.)

(1) Letting $\gamma(x)$ denote the γ code of 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; moreover, the padding bits are not accounted for as run-lengths.) 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 $|S_i|^{\mathbf{1}}$. 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 $|G_i|^{\mathbf{0}}$ (respectively, $|G_i|^{\mathbf{1}}$) denote the sum of run-lengths of **0**s (respectively, **1**s) relative to G_i. 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 analogously 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). Here we incur a logarithmic cost due to the binary search required in $A_{\mathbf{0}}$ or $A_{\mathbf{1}}$. All other work (accessing the array of pointers and traversing the bottom level) is constant-time.

The implementation of *rank* and *select* basically follows the same algorithmic structure. For example, executing $select_{\mathbf{1}}(x)$ starts out in the top level and performs a logarithmic binary search in $A_{\mathbf{1}}$ to find the position j of the predecessor $x' = A_{\mathbf{1}}[j]$ of x. (The interpolation search in this case does not help in practice to get $O(\log\log t)$ expected time.) Then, using the jth pointer, it accesses the byte-aligned codes for group G_j and scans G_j sequentially with partial sums looking at the $O(1)$ values $|S_i|^{\mathbf{0}}$ and $|S_i|^{\mathbf{1}}$ until it finds the position of the predecessor x'' for $x - x'$ inside G_j. At that point, a simple offset computation can lead to the suitable segment S_i (hence the reason for padding with dummy bits), whose $O(1)$ words are scanned sequentially for finding the predecessor of $x - x' - x''$ in S_i. We accumulate the partial sum of bits that are to the left of the latter 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.

As should be clear, the access is constant-time except for the binary search in $A_{\mathbf{0}}$ or $A_{\mathbf{1}}$. We will organize many of these directories into a tree of dictionaries (a *wavelet tree*), and thus perform a sequence of *select* operations along an upward traversal of p nodes/dictionaries through the tree. We reduce the $O(p \log t)$ cost to $O(p + \log t)$ by using an idea similar to fractional cascading [CG86], which we now describe briefly. Suppose dictionary D is the child of dictionary D' in the wavelet 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

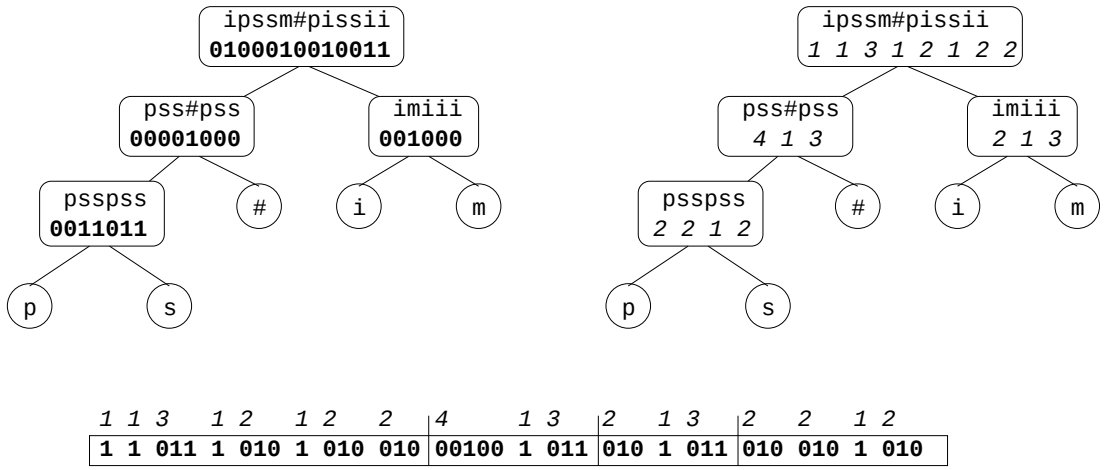

Figure 2: Left: an example wavelet tree. Right: an RLE encoding of the wavelet tree. Bottom: actual encoding on disk of right tree in heap layout with γ encoding.

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. This approach requires an additional array of pointers for the shortcut links, though as we will show in Section 4.2, the additional space required is 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, Gupta, and Vitter [GGV03] introduce the *wavelet tree* for reducing the redundancy inherent in maintaining separate dictionaries for each symbol appearing in the text. In order 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. 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.

We implicitly consider each left branch to be associated with a **0** and each right branch to be associated with a **1**. Each internal node 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, whose leaves are `p` and `s`. The dictionary (leaving aside the leading **0**) indicates that a single `p` appears in the `bwt` string, followed by two `s`'s, and so on. 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 as previously described. 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. This functionality is also sufficient to support multikey *rank* and *select*, which we support for any symbol $c \in \Sigma$ (one leaf per symbol, but again, not stored). For further discussion of the wavelet tree, see [GGV03].

We introduce two improvements for further speeding up the wavelet tree—use of fractional cascading and adopting a Huffman prefix tree shape. First, we implement links to enable 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

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

Table 4: Effect on performance of wavelet tree using fractional cascading and/or a Huffman prefix tree shape. The columns for Huffman and Cascading indicate whether that technique was used in that row. The values in the table represent a ratio of performance normalized with the best case (lower numbers are better). We do not show the improvement in performance from "linear" dictionary arrangements to the wavelet tree method, since it gains a factor of 10–20 in English texts as expected.

tree is oblivious to its shape [GGV03]. (We defer the details of the proof in the interest of brevity, though the reader may be satisfied with the observation that the linear method of evaluating dictionaries is nothing more than a completely skewed wavelet tree.)

We performed experiments to verify the truth of this theoretical observation in practice. Briefly, we generated $10,000$ random wavelet trees and computed the space required for various data. Our experiments indicated that a Huffman tree shape was never more than 0.006 bps more than any of our random wavelet trees. That translates into a less than 0.1% improvement in the compression ratio with respect to the original data. Moreover, most generated trees (over 90%) were actually worse than our baseline Huffman arrangement, and did not justify the additional computation time. Closer inspection of the data did not yield any insights as to how to generate a space-optimizing tree, even with the use of heuristics; one could, however marginally, improve our space by generating trees until a satisfactory replacement is found. Nevertheless, the key point is that the theoretical bound seems quite stable in practice.

Since the shape does not affect the space required, we can shape it so that it minimizes 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 the wavelet tree (which again is not stored explicitly), for $c \in \Sigma$. We build an optimal Huffman prefix tree by using $f(c)$ as a measure 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, we require, on average, just $1 + \log \sum_x f(x)/f(c)$ calls to *rank* or *select* involving leaf c.

Lemma 1 *Given a distribution of the 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.*

We make the empirical assumption that $f(c)$ is the frequency of c in the text (but other metrics are equally suitable as per the lemma), 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 4. The table contains timings for decompressing the entire file in question using repeated calls to the wavelet tree. This 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. 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. Some experiments (summarized in Table 5) indicate its value as just a compression mechanism dependent on the Φ transformation (see [GGV03] for more detail, but equivalently the

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

Table 5: Comparison of wavelet tree compression to standard methods. The values in the table are in bits per symbol (bps).

bwt stream). (Structures supporting fast access are not included here.) In the table, wave refers to the wavelet tree built on the original text; arit refers to the arithmetic code [RL79]; bzip2 version 1.0.2 is the Unix implementation of block sorting based on the Burrows-Wheeler transform; gzip is version 1.3.5; lha is version 1.14i [lha]; vh1 is Karl Malbrain and David Scott's implementation of Jeff Vitter's dynamic Huffman codes; zip is version 2.3; and wzip is the wavelet tree built on the Φ function. Note that the wavelet tree outperforms most other methods when built on the Φ function.

3.1 Efficient Construction of the Wavelet Tree

The lesson learned in the previous section suggests that the wavelet tree, coupled with RLE and γ encoding, is a simple but effective mean for compressing the output of block-sorting transforms. (See Table 5 for experimental findings.) 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 and added to our format wzip detailed below. The header 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). With reference to Figure 2, recall that the nodes of the wavelet tree are stored in heap ordering. 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, taking them in heap ordering and concatenating their encoding. Note that the wavelet tree has σ implicit leaves and $\sigma - 1$ internal nodes with dictionaries (see Figure 2).

Note that we do not need to store the length of each encoding, as it is already implicitly encoded as follows. When processing, the encoding for the root node of the wavelet tree ends when the sum of the encoded RLEs equals n (which may be spread over several blocks). At this point, we also know the total number of **0**s and **1**s, plus the (dummy) leading **0**. The former must be the sum of the RLE values in the next dictionary (in the left child), and the latter the sum of the one after that (in the right child). We can go on recursively this way, down to the implicit leaves, from which we can even 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 Burrows-Wheeler transformation (the Φ function in [GGV03], and hereafter the bwt stream) 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 uniform 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, as shown in Figure 2. 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 for each node, processing could take up to $O(\log|\Sigma|)$ time, it requires $O(n\log|\Sigma|)$ in total. We describe a refinement of this construction method requiring 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 (for compressible text, for instance).

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 have to 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 for each symbol (as we described in Section 3). Instead, we exploit the consecutive runs of equal symbols c, say r_c in number, in the input to avoid multiple passes. We then extend the runs by r_c units at a time. Let's define n_r as the number of such runs we will have to process for the entire `bwt` stream. Given any internal node in the tree, the set of values stored there are produced in increasing order, without explicitly creating the corresponding bitvector.

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 do not need to be 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`$[u]$, which is **1** if and only if we are encoding a run of **1**s.

Below, we describe 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). Here, we close the run in the current node u and start a new run (with cumulative run-length equal to zero) 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 the 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 run traversals of $O(\log|\Sigma|)$ length in the wavelet tree. (Recall that n_r is simply the number of runs processed by the algorithm.) It turns out

that $n_r = O(\min(n, nH_h))$, which proves our bound. Since $n_r \leq n$ trivially, we focus on showing 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 in 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 also in the `bwt` stream, except possibly for the first or the last run). Now, the number of runs in the dictionaries is upper-bounded by the sum of the logarithm of their run-lengths, which can be shown to be $O(nH_h)$ as in [GGV03].

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.

In order to facilitate this process, 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 ($u = 1$). We perform the downward traversal (lines 2–3), 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, as previously mentioned. We then find the rank of the symbol to decode. 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 is similar to lines 4–7 in `bwt2wzip`, except that we decrease the RLE values in the dictionaries (lines 5–7). The time required to decompress follows the same argument as for compressing.

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), XEO = Intel Xeon 2GHz 2GB Linux, gcc version 3.3.1 20030626 (Debian prerelease). We drew our data from the Canterbury and Calgary corpuses.

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

Table 6: Running times for `bwt2wzip` and `wzip2bwt` normalized with that of a simple copy routine.

The first three rows of Table 6 are files from those corpora; the last two rows are the concatenation of all the files in the same.

We compare our performance with a simple routine that copies the input `bwt` stream into another array. We normalize our routines with respect to this simple copy operation. We don't use the scan operation, as the compiler often cheats and doesn't actually generate code to scan if nothing happens. In these cases, "scan" is extremely fast, but misleading with regard to our experimental results. In our experiments, `bwt2wzip` (compression) is just 2—6 times slower than a simple copy operation, and `wzip2bwt` (decompression) is 3—7 times slower than the same. The difference in performance depends mainly on the architecture of the processor rather than the input file. (Consult Table 6 for proof of this fact, with bold figures for the minimum and the maximum.) The computation of RLE takes roughly 30% of the total time in `bwt2wzip` and 40% in `wzip2bwt`.

With regard to fine tuning performance in the code for `bwt2wzip` and `wzip2bwt`, each time we access an entry pointed by `dict`$[u]$, we may initiate a cache miss. Also, we need to pre-allocate more space than needed to accommodate all the dictionaries (whose final size is known at the end of the compression, which is too late). We can 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 the compression `bwt2wzip`. Indeed, during the computation, `wzip2bwt` accesses new RLE values in some nodes along a path in the wavelet only during the upward traversal (line 7). 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 for keeping all the RLE values that we will encode. 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 mentioned in Section 4.

4 Exploiting Suffix Arrays: Indexing Equals Compression

In Section 2, we explored dictionary methods which perform well in practice. Now, we apply these dictionary methods to compressed suffix arrays [GGV03, GV00, Sad00, Sad02b] and show both experimental success as well as a theoretical analysis of these practical methods. We refer the reader to [GGV03] for the background notions. We begin by proving the following theorem.

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 dictionary D using RLE+γ encoding.*

Proof: We note that each of our dictionaries D takes at most $2E(G)$ bits, which are bounded by $2\log\binom{n}{t}$ by Fact 1. We can replace our dictionaries with the ones in the analysis in [GGV03], at most doubling the theoretical worst-case bounds. The result follows automatically from the analysis in [GGV03]. □

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

We propose operating implicitly on *any* context order $h \geq 0$, and we argue that due to the nature of our directory, we are still able to achieve *simultaneously* the higher-order entropy given in [GGV03]. Moreover, we automatically balance the additional cost of retaining long contexts with respect to its impact on secondary structures. Said more mathematically, we can split the cost in [GGV03] as $nH_h + M(h)$, where $M(h)$ refers to the overhead necessary to encode a statistical model for contexts of length h. In other papers [FM04, GS03, Man01], it is assumed that $M(h)$ is a constant bounded by $O(|\Sigma|^h)$. However, this assumption fails for sufficiently large values in our experiments ($h \geq 4$). In fact, it is trivial to show that for sufficiently large h, we have $nH_h = 0$.

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

Proof: Build a suffix tree on the text terminated with n endmarkers which 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 (though not necessarily as extreme), the high-order entropy only has an asymptotic effect, whereas 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. Apparently, this issue has been somehow obscured in previous literature.

Now we prove our main theorem in this section, which describes how to encode the Φ function from [GGV03]. The neighbor function Φ is nothing more than the inverse of the LF mapping from the Burrows-Wheeler transform. It encodes for each position, in terms of suffix arrays, the location of the next suffix of the text in the suffix array.

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

Proof Sketch: In [GGV03], we conceptually break down the lists of the compressed suffix arrays into sublists for each context of order h (to scale the universe size in the dictionaries). We now are encoding all the sublists for the same symbol in one shot using our practical dictionaries (see also Section 3 for a review of wavelet trees). Hence, the difference in encoding is that we save space by not having to store pointers to the beginning of each sublist (which contribute significantly to the space $M(h)$ for the statistical model when there are many sublists). 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. Let's consider one such gap g, decomposed into three parts:

- g', the length of the jump out of the previous context (or sublist);
- g'', the length of the jump over empty contexts (or sublists) within a particular list;
- g''', the length of the jump from the beginning of the context (or sublist) containing this item.

The value g''' is exactly what is stored within a sublist if the context is explicit; our task remains to show that $\sum_{g \in G} \log g - \log g''' = o(n)$. Note also that since $\log g \leq \log(g' + g'') + \log g'''$ for all g, we note that encoding the three values together is strictly better than encoding the two pieces separately. In particular, the $\log(g' + g'')$ term is bounded by the pointer size to the sublists, and therefore we get the same space bound proven in [GGV03], with the exception that the coefficient

in front of the entropy term is 2 instead of 1 due to the γ encoding's coefficient of 2. (Note that δ coding could be used to achieve an even more succinct encoding—theoretically achieving a coefficient of 1—though we do not consider it due to its suboptimality in practice.)

We introduce notation to clarify the proof. Let the number of contexts be $c = \min\{|\Sigma|^h, n\}$, where $h \leq \alpha \log_{|\Sigma|} n$, for $0 < \alpha < 1$. In other words, $c \leq n^\alpha$. (This places the same restriction on the range for h as [GGV03]. See also Fact 3.) For each list, we can have at most c instances where we have non-zero values for g' and g''. Since the gaps of all such instances cannot exceed n, we can consider the worst case encoding for such a scenario – encoding c items equally out of n. Similar to arguments made previously, the bound then is $c \log(n/c) \leq n^\alpha (1-\alpha) \log(n) = o(n)$. Since this bound applies for each Σ list, we take at most $|\Sigma|$ times as much space, which is again, $o(n)$ (for compressible text), thus finishing the proof. □

One major advantage of block sorting is that not only does it compress according to high-order entropy, it also concisely represents the underlying statistical model. Ferragina and Manzini [FM00, FM01] employ a Move-to-Front (MTF) encoder [BSTW86] to capture the high-order entropy, but require a non-trivial representation of the model. In the next section, we describe how to use our dictionaries (RLE+γ), the suffix array (block sorting), and the wavelet tree (incremental representation of dictionaries) to achieve the same compression ratio of methods such as `bzip2`, without using MTF, arithmetic, or multi-table Huffman encoding. Giancarlo and Sciortino [GS03] also avoided using the MTF encoder, but it came at the price of a quadratic dynamic programming scheme. Ferragina and Manzini [FM04] recently devised a linear-time method to partition the `bwt` optimally for any given H_0 compressor, so as to achieve high-order entropy without using a MTF encoding. Moreover, finding a tighter encoding than γ for RLE (which is just as fast) would improve the state of the art on compressors.

4.1 Suffix Array Compression

Based on the experiments above, we can conclude that suffix arrays combined with the wavelet tree (described in Section 3) are the key to high-order compression. They avoid explicit treatment of the order of context, but allow for indirect context merging through the run-length encoding of the dictionaries employed in the wavelet tree. Our experiments also show that Move-To-Front (MTF) and Huffman/arithmetic coding are not strictly necessary to achieve high-order compression in our case. We detail these results in Table 7. Notice that Maniscalo and Golomb gain a huge savings from using MTF, but in all cases, γ performs better without. Indeed, γ is better than any other method for each file, aside from $E(L)$, which represents the lower bound on the specific code size we are using. Also note that values for `wave` from Table 5 are larger than their corresponding (non-MTF) entries in the γ column, as the former must includes some padding bits to allow fast access. In summary, we obtain high-order compression with three simple ingredients: suffix arrays, wavelet trees, and dictionaries based on RLE and γ encoding.

4.2 Suffix Array Functionalities

We now have all the ingredients for implementing compressed suffix arrays, which support the following operations.

Definition 1 Given a text T of length n, a *compressed suffix array* [GV00, Sad00, Sad02b] for T supports the following operations without requiring explicit storage of T or its (inverse) suffix array:

- *compress* produces a compressed representation that encodes (i) text T, (ii) its suffix array SA, and (iii) its inverse suffix array SA^{-1};
- *lookup* in SA returns the value of $SA[i]$, the position of the ith suffix in lexicographical order, for $1 \leq i \leq n$; *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 $1 \leq c \leq n - SA[i] + 1$.

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

Table 7: Measure of the effect of MTF on various coding methods when used with RLE. The MTF column indicates when it is used. The values in the table are in bits per symbol (bps).

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 8, we show the number of bits per symbol needed for compressed suffix arrays on some files from the Canterbury corpus. `ap90-64.txt` and `ap90-100.txt` represent concatenations of the first 64 MB (respectively, 100 MB) of news from Associated Press in 1990. Notice the minimal overhead cost for adding suffix array functionality. In addition, we compare our methods to those employed in the FM-index [FM00, FM01].

In the experiments we describe in Table 8, our Φ function is the functional equivalent of the tiny FM-index, and our compressed suffix array (CSA) is the functional equivalent of the fat FM-index. Note that our CSA always saves significant space over the fat FM-index, even when it is tuned specifically to support compression. (The split value in the entry for `bible.txt` indicates this specific tuning, though similar information was not available for other files.) Notice also that `gzip` and `bzip2` are compression tools *only*—they do not provide indexing at all. Yet, our CSA structure outperforms these methods on sufficiently large text (such as the last three columns for `bzip2` and almost across the board for `gzip`).

Note also 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. Our Φ function (compression of the `bwt` stream) performs to within 2% of the bounds obtained by the tiny FM-index, and performs better on `bible.txt`, `ap90-64.txt`, and `ap90-100.txt` (which are larger files).

5 Space-Efficient Suffix Trees

In this section, we apply our ideas on suffix arrays and compression to the implementation of a space-efficient version of suffix trees [Kur99]. We consider more than just the problem of searching, as suffix trees are at the heart of many algorithms on strings and sequences, so their full functionality is needed [Gus97]. 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 [Sad02a], which is significantly larger than that of the compressed suffix arrays discussed before. The bottleneck comes from retaining the longest common prefix (LCP) information, which requires at least $6n$ bits [Sad02b]. As an alternative, the same information can be maintained in at least $4n$ bits to retain the tree shape of at most $2n - 1$ nodes [MRS01], though there is a slowdown since the LCP information is not stored explicitly. In either case, a separate (compressed) suffix array is needed. As a result, the best theoretical representation of suffix trees can occupy more than $8n$ bits, which is the size of the text itself.

	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**
Tiny FM-index	-	1.687	2.154	1.570	1.771	-
Fat FM-index	-	1.911/2.582	2.689	2.658	2.839	-
gzip	3.264	2.352	2.312	2.344	3.032	3.000
bzip2	2.416	1.664	2.160	1.584	2.232	2.192

Table 8: Comparison of space required by Φ, the compressed suffix array (CSA), the FM-index, and standard compression algorithms, given in bits per symbol (bps). Overhead refers to all space other than the RLE+γ encoding for the data itself. Φ should be compared to the tiny FM-index, and the CSA to the fat FM-index. Most entries contain two values— the first is tuned for space, the second is tuned for speed. Singleton entries (for indexes) are the latter.

Note that the compressed suffix array encodes the leaves of the suffix tree. Since the 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 by the LCP is not drastically diminished, but it is expected since we are encoding the internal structure of the suffix tree.

To achieve less than $6n$ bits, we employed a simple heuristic that introduces an arbitrarily chosen parameter $S = O(\log n)$ that represents the slowdown factor. We implement part of the lowest common ancestor simplification introduced in [BFC00]. 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, we can 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`). 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 that must reside in external memory. There are several applications for which this is the case (e.g., storing the 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 [AKO02, AOK02, AKO04, AASA01]. For each node u, there are two integers $1 \leq u_l \leq u_r \leq n$ such that $SA[u_l \ldots u_r]$ contains all the suffixes stored in the leaves descending from u. Thus, $u \equiv (u_l, u_r, \ell_u)$ is a triple of integers in our representation, where ℓ_u represents the LCP between the strings of the text beginning at positions $SA[u_l]$ and $SA[u_r]$. In particular, for each node u, we 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 [Gus97].

We use Kasai et al. [KLea01] linear-time method to compute the LCP. We modified Sadakane's method [Sad02b] to store only LCP values larger than $2 \log n$; it works and compresses well. We also implement the doubling technique of Farach and Bender [BFC00] for computing the LCP in

constant time, though we can also trade time to reduce the space required.

We base our algorithms on the fact that, using LCP information, we can go from node u to node v by extending their intervals suitably and using the LCP 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 $lcp(SA[u'_l], SA[u'_r]) = \ell_v$. 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. 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 which we described in Section 3.4. The data sets used were drawn mainly from the Canterbury corpus `<http://corpus.canterbury.ac.nz>`. We also used Associated Press news `<http://trec.nist.gov/data/docs_eng.html>` 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, while module `csa` implements the compressed suffix array and related functionality (Section 4). Module `lcp` stores the 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 paper, we develop the simple notions of RLE and γ encoding to achieve competitive compression ratios and extremely fast compression and decompression time for both indexing and compression algorithms. Compared to inverted files, our text index requires 20% of the space of text, without requiring the text (self-indexing), and supporting faster, more powerful searches. We outperform the best known full-text indexing methods by roughly 10%, while still competing in search time (e.g., see [HLT$^+$04]). The techniques we have developed are practically sound, but also grounded in solid theoretical analysis and strong notions of encoding both the data and the underlying model. Our method is tuneable 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` outperforms `bzip2` in space and is competitive with `gzip` in time, though `gzip` is still faster. Our code does not require any additional parameters beyond the text size, alphabet size, and block size, and is tailored to work for large alphabets, e.g., Unicode, UTF/16. Our method performs integer bit assignments and does not resort to costly computation of fractional bits, as does an arithmetic coding technique. A simple copy operation is only 2–6 times faster than our `wzip` compression, and only 3–7 times faster than our decompression. As a matter of fact, our encoding algorithm is so fast that the true bottleneck is the encoding and decoding of γ! The main bottleneck remains the fast computation of the `bwt`.

However, despite these important 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 code is open source and easy to implement, as it uses introductory material on standard compression techniques. Our wavelet encoding is in some sense related to inversion coding [Deo02],

though the analysis in [GGV03] is the first to truly understand its impact. More critically, however, the wavelet tree serves as a vast improvement in access time over the inversion coding ideas. Other prefix codes (e.g., those in [Deo02, Fen96, Fen02, How97]) present other refinements with various tradeoffs which ultimately didn't yield improvements in practice. However, theoretical exploration of the suite of algorithms from [Deo02] could illuminate other approaches than the ones we have taken.

Both our compression and indexing methods depend directly upon the space bounds of our dictionaries; any improvement there yields significant savings on our method. The best possible compression achievable is that of $E(L)$; 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.

Acknowledgements

The authors would like to thank Rajeev Raman, Luca Foschini, and Raffaele Giancarl ofor 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

[AASA01] Hiroki Arimura, Hiroki Asaka, Hiroshi Sakamoto, and Setsuo Arikawa. Efficient discovery of proximity patterns with suffix arrays (extended abstract). In *CPM: 12th Symposium on Combinatorial Pattern Matching*, 2001.

[AKO02] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *WABI: International Workshop on Algorithms in Bioinformatics, WABI, LNCS*, 2002.

[AKO04] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[AOK02] Mohamed Ibrahim Abouelhoda, Enno Ohlebusch, and Stefan Kurtz. Optimal exact string matching based on suffix arrays. In *International Symposium on String Processing and Information Retrieval, SPIRE, LNCS*, volume 9, 2002.

[AUT] http://ccrma-www.stanford.edu/~jos/mdft/Autocorrelation.html.

[BFC00] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *Proceedings of LATIN*, LNCS, pages 88–94, 2000.

[BM99] Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM Journal on Computing*, 28(5):1627–1640, October 1999.

[BSTW86] J. Bentley, D. Sleator, R. Tarjan, and V. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, pages 320–330, 1986.

[Can] The Canterbury Corpus, `http://corpus.canterbury.ac.nz`.

[CG86] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

[Deo02] S. Deorowicz. Second step algorithms in the burrows-wheeler compression algorithm. In *Software–Practice and Experience*, volume 32, pages 99–111, 2002.

[Fen96] P. Fenwick. Punctured elias codes for variable-length coding of the integers. 1996.

[Fen02] P. Fenwick. Burrows-wheeler compression with variable-length integer codes. In *Software–Practice and Experience*, volume 32, pages 1307–1316, 2002.

[FM00] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science*, pages 390–398, 2000.

[FM01] Paolo Ferragina and Giovanni Manzini. An experimental study of an opportunistic index. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 269–278. ACM/SIAM, 2001.

[FM04] P. Ferragina and G. Manzini. Optimal compression boosting in optimal linear time using the burrows-wheeler transform. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2004.

[GBS92] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. New indices for text: PAT trees and PAT arrays. In *Information Retrieval: Data Structures And Algorithms*, chapter 5, pages 66–82. Prentice-Hall, 1992.

[GGV03] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Thitheenth Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM/SIAM, January 2003.

[GGV04] R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. January 2004.

[Goo] Google Inc., `http://www.google.com/help/refinesearch.html`.

[GS03] Raffaele Giancarlo and Marinella Sciortino. Optimal partitions of strings: A new class of Burrows-Wheeler compression algorithms. In *Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*. Springer, 2003.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[GV00] Roberto Grossi and Jeffrey S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proceedings of the Thirty-Second Annual ACM Symposium on the Theory of Computing*, pages 397–406, Portland, OR, 2000.

[HLT+04] W. Hon, T. Lam, W. Tse, C. Wong, and S. Yiu. Practical aspects of compressed suffix arrays and fm-index in searching dna sequences. In *6th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2004.

[How97] P. G. Howard. Interleaving entropy codes. In *Sequences*, 1997.

[HSS03] W. Hon, K. Sadakane, and W. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, pages 251–260, 2003.

[Jac89] Guy Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 549–554, 1989.

[KLea01] T. Kasai, G. Lee, and et al. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Combinatorial Pattern Matching (CPM)*, pages 181–192, 2001.

[Kur99] Stefan Kurtz. Reducing the Space Requirement of Suffix Trees. *Software – Practice and Experience*, 29(13):1149–1171, 1999.

[lha] `http://www.infor.kanazawa-it.ac.jp/ ishii/lhaunix/`.

[LV97] M. Li and P. Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer Verlag, 1997.

[Man01] Giovanni Manzini. An analysis of the Burrows — Wheeler transform. *Journal of the ACM*, 48(3):407–430, May 2001.

[McC76] Eduard M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[MM93] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[MNW95] A. Moffat, R. M. Neal, and I. H. Witten. Arithmetic coding revisited. *Data Compression Conference (DCC)*, 1995.

[MR97] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *38th Annual Symposium on Foundations of Computer Science*, pages 118–126, 1997.

[MRS01] J. Ian Munro, Venkatesh Raman, and S. Srinivasa Rao. Space efficient suffix trees. *Journal of Algorithms*, 39:205–222, 2001.

[MZ96] Alistair Moffat and Justin Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, October 1996.

[Nav] G. Navarro. The LZ-index: A Text Index Based on the Ziv-Lempel Trie. *TR DCC-2003-1. Department of Computer Science. University of Chile. Jan. 2003.*

[Nav04] G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms*, 2:87–114, 2004.

[Nel] Mark Nelson. Run length encoding/RLE. DataCompression.info, `http://www.datacompression.info/RLE.shtml`.

[NSN+00] Gonzalo Navarro, Edleno Silva de Moura, Marden Neubert, Nivio Ziviani, and Ricardo Baeza-Yates. Adding compression to block addressing inverted indexes. *Information Retrieval*, 3:49–77, 2000.

[Pag01] Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31:353–363, 2001.

[Rao02] S. Srinivasa Rao. Time-space trade-offs for compressed suffix arrays. *IPL*, 82(6):307–311, 2002.

[RL79] J. Rissanen and G.G. Langdon. Arithmetic coding. *IBM J. reserch and Develepment*, 23(2):149–162, March 1979.

[RRR02] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 233–242, 2002.

[Sad00] Kunihiko Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proceedings of ISAAC'00*, number 1969 in LNCS, pages 410–421, 2000.

[Sad02a] Kunihiko Sadakane, 2002. Personal Communication.

[Sad02b] Kunihiko Sadakane. Succinct representations of *lcp* information and improvements in the compressed suffix arrays. In *Proceedings of the Thitheenth Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM/SIAM, 2002.

[Sch] M. Schindler. http://www.compressconsult.com/rangecoder.

[WMB99] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, second edition, 1999.

[WZ99] Hugh E. Williams and Justin Zobel. Compressing integers for fast file access. *The Computer Journal*, 42(3):193–201, 1999.