# 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 similar to those in our earlier work [16], 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.

## 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 [18]. 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 [24, 38]. 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 [1]. An efficient combination of inverted file compression, block addressing, and sequential search on word-based Huffman compressed text is described in [27].

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 [21, 23] 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! 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 [15, 21] (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%).

Compressed suffix arrays [17, 32, 34, 36] and opportunistic FM-indexes [11, 12] represent new 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 [15], 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.

Grossi and Vitter [17] developed the compressed suffix array using $2n \log |\Sigma|$ bits in the worst case with $o(m)$ searching time. Sadakane [34, 36] 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 [11, 12], which is based on the Burrows-Wheeler transform (BWT) and is the first to encode the index size with respect to the $h$th-order empirical entropy $H_h$ of the text. Navarro [28] recently developed an index requiring $4nH_h + o(n)$ bits, and boasts fast search. Grossi, Gupta, and Vitter [16] exploited the higher-order entropy $H_h$ of the text to represent a compressed suffix array in just $nH_h + O(n \log \log n / \log_{|\Sigma|} n)$ 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

*Dipartimento di Informatica, Università di Pisa, via F. Buonarroti 2, 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 DAAD19–01–1–0725.

‡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 DAAD19–01–1–0725, by the National Science Foundation through grant CCR–9877133, and by an IBM research award.

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 provides $o(m)$ search time using only $o(n)$ bits. Several theoretical tradeoffs between space and search time were also developed [16].

In this paper, 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 [16] 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 $\gamma$ encoding. This shows that Move-to-Front (MTF) [8], arithmetic and Huffman encoding are not strictly necessary to achieve high-order compression with BWT. 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. We couple these tasks with an elegant analysis of high-order entropy using our simple encoding, as well as highlighting the importance of the underlying statistical model. 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.

## 2 A Simple Yet Powerful Dictionary

Succinct dictionaries [9, 30] are constant-time rank and select data structures occupying tiny space. They store $t$ entries chosen from a bounded universe $[0 \ldots n-1]$ (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 $[0 \ldots n-1]$. 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 $i$th **1** in $B$. (Analogous definitions hold for **0**s.) The $i$th bit in $B$ can be computed as $B[i] = rank_{\mathbf{1}}(i) - rank_{\mathbf{1}}(i-1)$. The constant-time fully indexable dictionaries [31] 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 the bitvector, and $p(\mathbf{0}) = 1 - p(\mathbf{1})$. Since

$$H_0 = -p(\mathbf{0}) \log p(\mathbf{0}) - p(\mathbf{1}) \log p(\mathbf{1}) \sim \log \binom{n}{t},$$

we can think of dictionaries as 0th order compressors which 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 [19, 25]. Recently, dictionaries have been shown to be crucial for text indexing data structures [16]. Specifically, the data structuring framework in [16] 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, 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 $H_0 \sim \log \binom{n}{t}$.

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 really 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 does not account for the distribution of the **1**s and **0**s inside $B_D$, which, in the examples above, is favorable for a better than arbitrary compression.

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 [38]. 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 $\mathbf{0}$ and $\mathbf{1}$.

The length $\ell$ is then encoded in some fashion. One such method is the $\gamma$ code, which represents the length $\ell$ 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. The $\delta$ code requires fewer bits asymptotically by encoding $1 + \lfloor \log \ell \rfloor$ via the $\gamma$ code rather than in unary, thus requiring $1 + \lfloor \log \ell \rfloor + 2\lfloor \log \log 2\ell \rfloor$ bits. 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 $\ell$. Byte-aligned code splits the $lb(\ell)$ bits into groups of 7 bits each, prepending a bit as most significant to indicate whether there are more bits of $\ell$ in the next byte. We refer to [38] 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 $\mathbf{1}$; if not, we can use an extra bit to denote this case and encode the final gap length separately. We also assume that $t \leq n/2$ or else we reverse the role of $\mathbf{0}$ and $\mathbf{1}$.

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 2.1. *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 $\mathbf{1}$s out of a universe of size $n$.*

*Proof.* Let's assume for the moment that $B_D$ ends with a $\mathbf{1}$. 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 $\mathbf{1}$, 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:

FACT 2.2. *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 $\mathbf{1}$s out of a universe of size $n$.*

| $\log(gap)$ | RLE+$\gamma$ | Gap+$\gamma$ | $\log \binom{n}{t}$ | $E(L)$ | $E(G)$ |
|---|---|---|---|---|---|
| 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+$\gamma$ codes, and $\log \binom{n}{t}$. Each bitvector $B_D$ is produced by choosing a maximum gap length and generating uniformly random gaps of $\mathbf{0}$s between consecutive $\mathbf{1}$s. The gap column indicates the maximum gap length on a logarithmic scale. The values in the table are the bits per gap required by each method.

*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 Kolmogorav complexity of an arbitrary string is undecidable. 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 $\mathbf{1}$ (or $\mathbf{0}$), that their cost is more than paid for by the improved coding of contiguous items. Notice also that RLE+$\gamma$ outperforms Gap+$\gamma$ for small gap sizes (namely 4 or less). This behavior motivates our choice for RLE, as many gap sizes are small in our distributions. The columns for RLE+$\gamma$ and Gap+$\gamma$ codes refer to taking $B_D$ and encoding each run-length (or gap) using the $\gamma$ code, rather than $lb$ (which is not a prefix code).

We also performed experiments comparing the space occupancy of several different encodings in place of the $\gamma$ 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 [29] 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 method shows that the underlying distribution of gaps in our data is Bernoulli. (When $b = 1$, the skewed Bernoulli code is equal to $\gamma$.) Notice that, except for `random.txt`, $\gamma$ codes are less than 1 bps from $E(L)$. For random

text, $\gamma$ codes are not as good as expected. Note also that $E(G)$ and Gap+$\gamma$ outperform their respective counterparts on `random.txt`, which represents the worst case for RLE.

We can use $\delta$ codes to store $B_D$, using just $E(L)+\sum_{i=1}^{j}\lfloor\log\log(2\ell_i)\rfloor$ bits by Fact 2.2. Similarly, $\gamma$ coding require $2E(L)-t$ bits, though in practice it outperforms $\delta$, since $\gamma$ is more efficient for small run-lengths. For a detailed study on the encoding of arbitrary integers sequences, we refer the reader to [37]. In this paper, we focus on encoding RLE values for the dictionary problem. Table 2 suggests $\gamma$ as best encoding to couple with RLE.

In order to support fast access, we can use a simple scheme from [9, 19, 30, 31] and pay an additional cost of $o(n)$ bits. In the full paper, we show how to support *rank* and *select* in constant time with our encoding. As previously noted, 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.

**2.2 Fast Access of Practical Dictionaries** We focus here 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 structures 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 bitvector $B_D$:

(1) Letting $\gamma(x)$ denote the $\gamma$ 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 with each segment dummy **1**s so they all have the same length of $O(1)$ words (this padding add a total number of bits which is negligible; moreover, the padding bits are not accounted 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 $|S_i|^0$ and $|S_i|^1$ for each segment $S_i$, where $|S_i|^0$ (resp., $|S_i|^1$) denotes the sum of run-lengths of **0**s (resp., **1**s) relative to $S_i$. We store each value of the sequence $|S_1|^0,|S_1|^1,|S_2|^0,|S_2|^1,\ldots,|S_k|^0,|S_k|^1$ using the byte-aligned codes with continuation bit. We then divide the resulting encoded sequence into groups $G_1,G_2,\ldots,G_m$, each group containing several values of $|S_i|^0$ and $|S_i|^1$. 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 $|G_i|^0$ (resp., $|G_i|^1$) denote the sum of run-lengths of **0**s (resp., **1**s) relative to $G_i$. The $i$th entry of $A_{\mathbf{0}}$ stores the prefix sum $\sum_{l=1}^{i}|G_l|^0$. The entries of $A_{\mathbf{1}}$ are analogously defined. We also keep an array of pointers, where the $i$th pointer refers to the starting position of $G_i$ in byte-aligned encoding in 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 essentially constant-time.

The implementation of *rank* and *select* follows essentially 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 $j$th pointer, it accesses the byte-aligned codes for group $G_j$ and scan sequentially $G_j$ with partial sums looking at the $O(1)$ values $|S_i|^0$ and $|S_i|^1$ 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 $S_i$ (this is why they are padded 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 it 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 performing 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 [10]. Suppose a dictionary $D$ is the child of dictionary $D'$ in the wavelet tree. Suppose 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. This approach requires an additional array of pointers for the shortcut links.

## 3 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 [16, 17, 34, 36] and show both experimental success as well as a theoretical analysis of these practical methods. We

| File | $E(L)$ | $E(G)$ | RLE+$\gamma$ | Gap+$\gamma$ | RLE+$\delta$ | 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. Unless stated otherwise, the listed coding method is used with RLE. The files indicated are from the Canterbury Corpus [2]. The values in the table are the bps required by each method.

refer the reader to [16] for the background notions. We begin by proving the following theorem.

THEOREM 3.1. *We can encode the $n_k$ entries in all sublists at level $k$ of the compressed suffix array using at most $2nH_h + n_k \log e + o(n)$ if we store each sublist as a dictionary $D$ using RLE+$\gamma$.*

*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 2.1. We can replace our dictionaries in the analysis by Lemma 3.6 in [16], at most doubling the theoretical worst-case bounds. The result follows automatically. □

This discovery brings up a remarkable point—our practical dictionary is blind to the universe size that was so carefully constructed in [16] to allow the use of the fully indexable dictionaries from [31] (whose space occupancy is almost linearly dependent with 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 [16]. Moreover, we automatically balance the additional cost of retaining long contexts versus its impact on secondary structures. Said more mathematically, we can split the cost in [16] 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 [14, 22], 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 $nH_h = 0$. 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 $\Phi$ function from [17]. The neighbor function $\Phi$ 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 3.2. *We can encode $\Phi$ using $2nH_h + n + o(n)$ bits with $\gamma$ encoding, thus implicitly achieving high-order entropy.*

*Proof Sketch*: In [16], 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 incrementally (see also Section 3.1). Hence, the difference in encoding is that we save 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 traverses a sublist. The idea 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 [16], with the exception that the coefficient in front of the entropy term is 2 instead of 1 due to the $\gamma$ encoding's coefficient of 2. (Note that $\delta$ 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 from [16] 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 [16].) 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 $\Sigma$ 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 [11, 12] employ a Move-to-Front (MTF) encoder [8] 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+$\gamma$), the suffix array (block sorting), and the wavelet tree (incremental representation of lists) to achieve the same compression ratio of methods as `bzip2`, without using MTF, arithmetic, or multi-table Huffman encoding. Giancarlo and Sciortino [14] also avoided using the MTF encoder, but it came at the price of a quadratic dynamic programming scheme. Ferragina and Manzini [13] recently devised a linear-time method to partition 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 $\gamma$ for RLE would improve the state of the art on compressors.

**3.1 Wavelet Trees** Grossi, Gupta, and Vitter [16] describe a method 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. For instance, list 1 encodes the locations of elements in its list relative to all other lists. For those locations not used by list 1, list 2 encodes the locations of elements in list 2 relative to lists 3, 4, .... For those locations not taken up by lists 1 or 2, list 3 gives the locations of elements in list 3 relative to lists 4, 5, ..., etc.

Encoding the lists this way achieves the high-order entropy, as per the discussion in Lemma 4.1 of [16]. However, the lookup time for a particular item is now linear in the number of lists, as a query must backtrack through all the previous lists to reconstruct the answer. The *wavelet tree* relates a list to an exponentially growing number of lists, rather than simply all prior encoded lists.

Consider a completely balanced wavelet tree, where we do not actually store the leaves (i.e. the actual sublists themselves). We implicitly consider each left branch to be associated with a **0** and each right branch to be associated with a **1**. Each internal node is a dictionary $D$ with the elements in its left subtree stored as **0**, and the elements in its right subtree stored as **1**. For instance, the root node of the tree in Figure 1 represents each of the positions of $s_1, \ldots, s_4$ as a **0**, and each of the positions of $s_5, \ldots, s_8$ as a **1**.

Since there are at most $|\Sigma|$ lists, any symbol from the text can be decoded in just $O(\log |\Sigma|)$ time. This

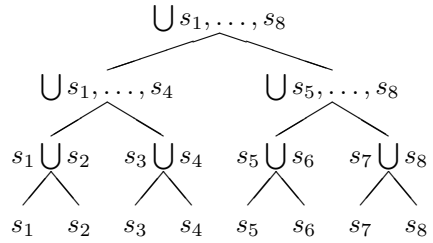


Figure 1: A wavelet tree

functionality is also sufficient to support multikey *rank* and *select*, in which we support them for any symbol $c \in \Sigma$ (one leaf per symbol, but again, not stored). For further discussion of the wavelet tree, see Section 4.2 in [16].

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 2.1. Second, one can prove that theoretically, the space occupancy of the wavelet tree is oblivious to its shape. (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 lists is nothing more than a completely skewed wavelet tree.)

We performed experiments to verify the truth of this 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 under the assumption that the distribution of the calls to the operations in 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, for $c \in \Sigma$. Let's 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 $c$.

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

Table 3: 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 lists to the wavelet tree as it gains a factor of 10–20 in English texts as expected.

LEMMA 3.1. *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 the 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), 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. The results are summarized in Table 3. 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 wavelet tree can be built in $O(nH_0)$ time as follows. For each leaf $l$ (considered in left to right order), propagate its entries to its ancestors. If $l$ is a left child, stop the current run-length of **1**s (if any), otherwise increment the number of **0**s in the current run-length; if $l$ is a right child, perform the symmetric operations. Given any internal node in the tree, the set of values stored there are produced in increasing order, without explicitly creating the corresponding bitvector.

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 4) indicate its value as a compression mechanism dependent on the $\Phi$ transformation of the data. In the table, `wave` refers to the wavelet tree built on the original text; `arit` refers to the arithmetic code [33]; `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 [3]; `vh1` is Karl Malbrain and David Scott's implementation of Vitter's dynamic Huffman codes; `zip` is version 2.3; and `wavesa` is the wavelet tree built on the $\Phi$ function. Note that the wavelet tree outperforms most other methods when built on the $\Phi$ function.

**3.2 Suffix Array Compression** Based on the experiments above, we can conclude that suffix arrays combined with the wavelet tree are the key to high-order compression. They avoid explicit treatment of the order of context (unlike [16]), 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 5. Notice that Maniscalo and Golomb gain a huge savings from using MTF, but in all cases, $\gamma$ performs better without. Indeed, $\gamma$ 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 `wavesa` from Table 4 are larger than their corresponding (non-MTF) entries in the $\gamma$ 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 $\gamma$ encoding. Interestingly enough, wavelet trees coupled with suffix arrays are multi-purpose data structures – they can implement the $\Phi$ function of compressed suffix arrays (using multikey *select*) or the $LF$ mapping of the Burrows-Wheeler transform in the FM-index (using multikey *rank*).

**3.3 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* [17, 34, 36] 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 $i$th 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 $j$th 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 $1 \leq c \leq n - SA[i] + 1$.

We still need to store $SA_\ell$ and its inverse, as well as a dictionary to mark the positions in the original suffix array represented in $SA_\ell$. Here we

| File | wave | arit | bzip2 | gzip | lha | vh1 | zip | **wavesa** |
|---|---|---|---|---|---|---|---|---|
| book1 | 5.335 | 4.530 | 2.992 | 2.953 | 2.967 | 4.563 | 2.954 | **2.619** |
| E.coli | 2.248 | 2.008 | 2.189 | 2.337 | 2.240 | 2.246 | 2.337 | **2.181** |
| bible.txt | 5.004 | 4.309 | 1.931 | 1.941 | 1.939 | 4.353 | 1.941 | **1.631** |
| 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 4: Comparison of wavelet tree compression to standard methods. The values in the table are in bps.

| File | MTF | $E(L)$ | $\gamma$ | $\delta$ | 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 5: 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 bps.

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 6, we show the number of bits per symbol needed for compressed suffix arrays on some files from the Canterbury corpus. Notice the minimal overhead cost for adding suffix array functionality. In addition, we compare our methods to those employed in the FM-index [11, 12].

In the experiments we describe in Table 6 our $\Phi$ 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.) 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 $\Phi$ function performs to within 2% of the bounds obtained by the tiny FM-index, and performs better on `bible.txt` and `ap90-64.txt`.

## 4 An Application: Space-Efficient Suffix Trees

In this section, we apply our ideas to the implementation of a space-efficient version of suffix trees [20]. 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 [18]. 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 [35], 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 [36]. 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 [26], 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.

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 [7]. 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`),

| | book1 | bible.txt | E.coli | world192.txt | ap90-64.txt | ap90-100.txt |
|---|---|---|---|---|---|---|
| $\Phi$ 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 |
| $\Phi$ | 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 | - |

Table 6: Comparison of space required by $\Phi$, the compressed suffix array (CSA), and the FM-index, given in bps. Overhead refers to all space other than the RLE+$\gamma$ encoding for the data itself. $\Phi$ 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 are the latter.

2.54 bps (`E.coli`), 2.21 bps (`bible.txt`) and 2.8 bps (`world192.txt`). These sizes are comparable to those obtained by `gzip`, namely, 3.26 bps (`book1`), 2.31 bps (`E.coli`), 2.35 bps (`bible.txt`) 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, 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 also used recently to devise efficient algorithms [4, 5, 6]. 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 $\ell_u$ 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 length of 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 [18].

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 details for most operations until the full version of this paper, 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 leftward and the other for $u'_r$ going rightward. At each step of our binary search in position $i$, we compute $lcp(SA[i], SA[u'_l])$ and compare it with $\ell_v$. 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.

## 5 Implementation Details

In this section, we discuss our experimental setup. Many experiments were run on an IBM xSeries 335 Server. This machine has a 2.0 GHz Intel Xeon processor with a 512 KB L2 cache, 2.0 GB of PC2100 DDR-SD RAM, and a 40 GB IDE hard drive with 2 MB cache. We also employed a 1.0 GHz Athlon based PC with 512MB RAM. Both machines are running Debian Linux 3.0, with a `gnu gcc/g++ 3.2` compiler. 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 3). Module `lcp` stores the LCP information and module `st` implements suffix tree functionality, though we avoid storing any nodes explicitly (Section 4). 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

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%.

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 cost. Our method is tuneable to the access pattern of any file, which is a property unknown in similar work on compressed indexing. We construct the index in competitive time (roughly 1-2 minutes for 64 MB of data on our test system). We plan to perform intense algorithm engineering to further tune the search time of our structures. Despite these achievements, we believe that our space can be further reduced by considering more sophisticated codes for integers, such as practical arithmetic coding or `szip`. Our method directly depends upon the space bounds of our dictionaries; any improvement there yields significant savings on our method. The best possible compression achievable is that of $E(L)$; there is significant room for improvement. Our key is to exploit the underlying entropy in the text using a transform and a solid method of removing redundancy using the wavelet tree.

## References


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

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

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

[4] M.I. Abouelhoda, S. Kurtz, E. Ohlebusch. The enhanced suffix array its applications to genome analysis. WABI 2002.

[5] M.I. Abouelhoda, E. Ohlebusch, S. Kurtz. Optimal exact string matching based on suffix arrays. SPIRE 2002.

[6] H. Arimura, H. Asaka, H. Sakamoto, S. Arikawa. Efficient discovery of proximity patterns with suffix arrays (extended abstract). CPM 2001.

[7] M.A. Bender M. Farach-Colton. The LCA problem revisited. LATIN 2000.

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

[9] A. Brodnik, J. Munro. Membership in Constant Time and Almost-Minimum Space. SIAM Journal on Computing, 5:1627–1640, 1999.

[10] B. Chazelle, L.J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.

[11] P. Ferragina, G. Manzini. Opportunistic data structures with applications. FOCS 2000.

[12] P. Ferragina, G. Manzini. An experimental study of an opportunistic index. SODA 2001.

[13] P. Ferragina, G. Manzini. Optimal Compression Boosting in Optimal Linear Time using the Burrows-Wheeler Transform. SODA 2004.

[14] R. Giancarlo, M. Sciortino. Optimal partitions of strings: A new class of Burrows-Wheeler compression algorithms. CPM 2003.

[15] G.H. Gonnet, R.A. Baeza-Yates, T. Snider. New indices for text: PAT trees PAT arrays. *Information Retrieval: Data Structures Algorithms*, 1992.

[16] R. Grossi, A. Gupta, J. S. Vitter. High-order entropy-compressed text indexes. SODA 2003.

[17] R. Grossi, J.S. Vitter. Compressed suffix arrays suffix trees with applications to text indexing string matching (extended abstract). STOC 2000.

[18] Dan Gusfield. *Algorithms on Strings, Trees Sequences: Computer Science Computational Biology.*, 1997.

[19] G. Jacobson. Space-efficient static trees graphs. FOCS 1989.

[20] S. Kurtz. Reducing the Space Requirement of Suffix Trees. *Software–Practice Experience*, 29:1149–1171, 1999.

[21] U. Manber, G. Myers. Suffix arrays: a new method for on-line string searches. *SICOMP*, 22(5):935–948, 1993.

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

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

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

[25] J. I. Munro, V. Raman. Succinct representation of balanced parentheses, static trees planar graphs. FOCS 1997.

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

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

[28] G. Navarro. The LZ-index: A Text Index Based on the Ziv-Lempel Trie. Manuscript.

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

[30] R. Pagh. Low redundancy in static dictionaries with constant query time. *SICOMP*, 31:353–363, 2001.

[31] R. Raman, V. Raman, S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding $k$-ary trees multisets. SODA 2002.

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

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

[34] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. ISAAC 2000.

[35] K. Sadakane, 2002. Personal Communication.

[36] K. Sadakane. Succinct representations of *lcp* information improvements in the compressed suffix arrays. SODA 2002.

[37] H.E. Williams, J. Zobel. Compressing integers for fast file access. *The Computer Journal*, 42(3):193–201, 1999.

[38] I.H. Witten, A.Moffat, T.C. Bell. *Managing Gigabytes: Compressing Indexing Documents and Images*, 1999.

@article{Grossi2004WhenIE, title={When indexing equals compression: experiments with compressing suffix arrays and applications}, author={Roberto Grossi and Ankur Gupta and Jeffrey Scott Vitter}, journal={ACM Trans. Algorithms}, year={2004}, volume={2}, pages={611-639} }. Roberto Grossi, Ankur Gupta, Jeffrey Scott Vitter. Published 2004. Computer Science. ACM Trans. Algorithms. When indexing equals compression: Experiments on compressing suffix arrays and applications. In Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms. ACM-SIAM Press, New York, 636--645.]] Google Scholar Digital Library. Grossi, R., and Vitter, J. 2000. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In Proceedings of the 32nd ACM Symposium on Theory of Computing. ACM, New York, 397--406.]] But again broadly speaking, compressing the suffix array will cause the search for a pattern of length m (which can be O(m) in an uncompressed suffix array, if carefully implemented) to be delayed by a factor that depends (usually logarithmically) on the length of the entire text. Furthermore, any approach making use of wavelet trees means an additional dependence on the size of the alphabet. Index Key Compression allows us to compress portions of the key values in an index segment (or Index Organized Table), by reducing the storage inefficiencies of storing repeating values. It compresses the data by splitting the index key into two parts; the leading group of columns, called the prefix entry (which are potentially shared across multiple key values), and the suffix columns (which is unique to every index key).Â  Suffix entries form the compressed representation of the index key. Each one of these compressed rows refers to the corresponding prefix, which is stored in the same block. Compressed suffix arrays are a general class of data structure that improve on the suffix array.[1][2] These data structures enable quick search for an arbitrary string with a comparatively small index. Given a text T of n characters from an alphabet Î£, a compressed suffix array supports searching for arbitrary patterns in T. For an input pattern P of m characters, the search time is typically O(m) or O(m + log(n)). The space used is typically. O ( n H k ( T ) ) + o ( n ) {\displaystyle O(nH_{k}(T))+o(n)}. , where. H k ( T ) {\displaystyle H_{k}(T)}.Â  ^ L. Foschini, R. Grossi, A. Gupta, and J. S. Vitter, Indexing Equals Compression: Experiments on Suffix Arrays and Trees, ACM Transactions on Algorithms, 2(4), 2006, 611-639.