Wavelet Tree, Part II: Text Indexing

Paolo Ferragina
Department L'EMbeDS, Sant'Anna School of Advanced Studies, Pisa, Italy
Department of Computer Science, University of Pisa, Italy

Raffaele Giancarlo
Department of Mathematics and Computer Science, University of Palermo, Italy

Roberto Grossi
Department of Computer Science, University of Pisa, Italy

Giovanna Rosone
Department of Computer Science, University of Pisa, Italy

Rossano Venturini
Department of Computer Science, University of Pisa, Italy

Jeffrey Scott Vitter
Department of Computer Science, Tulane University, New Orleans, LA, USA
Department of Computer and Information Science, The University of Mississippi, MS, USA

Abstract

The *Wavelet Tree* data structure introduced in Grossi, Gupta, and Vitter [11] is a space-efficient technique for rank and select queries that generalizes from binary symbols to an arbitrary multisymbol alphabet. Over the last two decades, it has become a pivotal tool in modern full-text indexing and data compression because of its properties and capabilities in compressing and indexing data, with many applications to information retrieval, genome analysis, data mining, and web search. In this paper, we survey the fascinating history and impact of Wavelet Trees; no doubt many more developments are yet to come. Our survey borrows some content from the authors' earlier works.

This paper is divided into two parts: The first part gives a brief history of Wavelet Trees, including its varieties and practical implementations, which appears in the Festschrift dedicated to Roberto Grossi [4]; the second part (this one) deals with Wavelet Tree-based text indexing and is included in the Festschrift dedicated to Giovanni Manzini.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Theory of computation → Data compression

Keywords and phrases Wavelet tree, data compression, text indexing, compressed suffix array, Burrows-Wheeler transform, rank and select

Digital Object Identifier 10.4230/OASIcs.Manzini.2025.4

Acknowledgements The authors would like to thank Hongwei Huo for helpful comments and figures.

1 Introduction

The field of compressed full-text indexing [24] involves the design of data structures (aka, indexes) that support fast substring matching using small amounts of space. For a text string $\mathcal{T}[1,n]$ over an arbitrary alphabet Σ of size σ and a pattern $\mathcal{P}[1,m]$, the goal of text indexing is to preprocess $\mathcal{T}$ using succinct space so that queries like the following can be quickly answered: (1) count the number *occ* of occurrences of $\mathcal{P}$ in $\mathcal{T}$; (2) locate the *occ* positions in $\mathcal{T}$ where $\mathcal{P}$ occurs; and (3) starting at text position *start*, extract the length-ℓ substring $\mathcal{T}[\mathit{start}, \mathit{start}+\ell-1]$.

A main goal is to create an index whose size is roughly equal to the size of the text in compressed format, with search performance comparable to the well-known indexes on uncompressed text, such as suffix trees and suffix arrays. Some compressed data structures are in addition *self-indexes* in that the data structure encapsulates the original text and, thus, can quickly recreate any portion of it. As a result, the original text is not needed, it can be discarded and replaced by only the (self-)index.

Most compressed indexing techniques developed in the last two decades make use of the powerful *Wavelet Tree* data structure, developed by Grossi, Gupta, and Vitter [11], with many applications to information retrieval, genome analysis, data mining, and web search. The Wavelet Tree supports fast rank and select queries on any string from an arbitrary alphabet. As such, it provides a space-efficient extension of the space-efficient rank-select data structures for binary sequences [26, 24] to the case of general alphabets of arbitrary size.

A Wavelet Tree represents any string from a general multisymbol alphabet as a hierarchical set of binary strings. It has the following key property for purposes of compression: If each such binary string is encoded according to its information-theoretic minimum size, then the original string is compressed to its information-theoretic minimum size.

In this paper we discuss how Wavelet Trees can be used to construct an efficient self-index for text $\mathcal{T}$ using space related to the higher-order entropy of $\mathcal{T}$. The two main data structures to which Wavelet Trees are applied for that purpose are the compressed suffix array (CSA) [12, 28, 11] and the FM-index [6, 7].

2 Preliminaries

We refer the reader to `PART I`, which is included in the Festschrift dedicated to Roberto Grossi [4], for the notation used in this paper, the history and background of the ubiquitous Wavelet Tree, and the key definitions of 0th order entropy and higher-order entropy.

The suffix array SA gives the positions of all the suffixes of $\mathcal{T}$ in lexicographical order. We use $SA[0, n-1]$ to denote the suffix array (SA) of $\mathcal{T}$. We use $SA^{-1}[0, n-1]$ to denote the inverse suffix array, where SA^{-1} is a permutation of suffixes of $\mathcal{T}$ in lexicographical order, such that $SA^{-1}[SA[i]] = i$. Table 1 provides a running example.

The key notion of *neighbor function* Ψ [12, 28] is illustrated in Table 1 and formally defined as follows: Given a suffix s of $\mathcal{T}$ and its index i in the suffix array (i.e., $s = \langle \mathcal{T}[SA[i]], \mathcal{T}[SA[i]{+}1 \bmod n], \dots \rangle$), let s' be the suffix (a.k.a. neighbour) formed by removing the first symbol of s. The value of the neighbor function $\Psi[i]$ is the index i' of s' in the suffix array. More formally,

$$\Psi(i) = SA^{-1}[(SA[i] + 1) \bmod n].$$

Let us designate by $\mathcal{T}_{\mathsf{x}}$ (as in `PART I`) the substring of $\mathcal{T}$ associated with an individual context x of k symbols. Wavelet Trees are a natural and elegant way to compress each substring $\mathcal{T}_{\mathsf{x}}$ so that the code length per symbol is the 0th-order entropy of $\mathcal{T}_{\mathsf{x}}$. It thus follows by Definition 3 in `PART I` of kth-order entropy that the cumulative encoding of all the substrings $\mathcal{T}_{\mathsf{x}}$ achieves kth-order entropy of the full string $\mathcal{T}$. We elaborate on this key idea in Section 4.

3 Text Indexing

The two main paradigms used for modern text indexing are compressed suffix arrays (CSA) [12, 28, 11] and FM-index [6, 7]. We use the notation $\mathcal{T} = \mathcal{T}[0, n-1]$ to denote a text of n symbols drawn from an alphabet Σ of size σ. Both the CSA and FM-Index make use of the suffix array concept.

Table 1 For text $\mathcal{T}$ = `tcaaaatatatgcaacatatagtattagattgtat`# shown in the first column, the subsequent columns show the suffix array SA, the inverse suffix array, the neighbor function Ψ, and the LF function of the BWT. The second-to-last column shows the suffixes in sorted order; each suffix starts at the symbol under F (first) and ends at the symbol L (last). The string of symbols under L is the BWT of $\mathcal{T}$. The last column $\mathcal{B}$ is the bit array for the root node of the Wavelet Tree of L; symbols #, `a`, and `c` (for the left subtree) are designated by a 0, and symbols `g` and `t` (for the right subtree) are designated by a 1. The Wavelet Tree of L is pictured in Figure 1.

i	$\mathcal{T}$	SA	SA^{-1}	Ψ	LF	F	L	$\mathcal{B}$
0	t	35	31	31	23	# t c a . . . a	t	1
1	c	2	16	2	16	a a a a . . . t	c	0
2	a	3	1	4	1	a a a t . . . c	a	0
3	a	13	2	5	17	a a c a . . . g	c	0
4	a	4	4	11	2	a a t a . . . a	a	0
5	a	14	11	18	3	a c a t . . . c	a	0
6	t	26	28	19	24	a g a t . . . t	t	1
7	a	20	12	22	25	a g t a . . . a	t	1
8	t	33	29	23	26	a t # t . . . g	t	1
9	a	18	13	25	27	a t a g . . . a	t	1
10	t	16	32	27	18	a t a t . . . a	c	0
11	g	5	20	28	4	a t a t . . . a	a	0
12	c	7	17	29	28	a t a t . . . a	t	1
13	a	9	3	32	29	a t g c . . . a	t	1
14	a	23	5	34	30	a t t a . . . g	t	1
15	c	28	18	35	19	a t t g . . . a	g	1
16	a	1	10	1	31	c a a a . . . #	t	1
17	t	12	27	3	20	c a a c . . . t	g	1
18	a	15	9	10	5	c a t a . . . a	a	0
19	t	27	25	15	6	g a t t . . . t	a	0
20	a	11	7	17	32	g c a a . . . a	t	1
21	g	31	22	26	33	g t a t . . . t	t	1
22	t	21	30	30	7	g t a t . . . t	a	0
23	a	34	14	0	8	t # t c . . . t	a	0
24	t	25	34	6	34	t a g a . . . a	t	1
25	t	19	24	7	9	t a g t . . . t	a	0
26	a	32	6	8	21	t a t # . . . t	g	1
27	g	17	19	9	10	t a t a . . . c	a	0
28	a	6	15	12	11	t a t a . . . a	a	0
29	t	8	35	13	12	t a t g . . . t	a	0
30	t	22	33	14	22	t a t t . . . a	g	1
31	g	0	21	16	0	t c a a . . . t	#	0
32	t	10	26	20	13	t g c a . . . t	a	0
33	a	30	8	21	35	t g t a . . . a	t	1
34	t	24	23	24	14	t t a g . . . t	a	0
35	#	29	0	33	15	t t g t . . . g	a	0

3.1 Compressed Suffix Array (CSA)

The CSA represents the text string $\mathcal{T}$ by encoding the *neighbor function* Ψ [12, 28]. The sequence of Ψ values suffices to recreate the original text $\mathcal{T}$, using some supporting data structures. For example, if we know the symbol that appears at the beginning of the ith suffix in lexicographic order, we can easily determine the neighboring symbol in the text (at position $SA[i]+1$ in the text) by computing $j = \Psi(i)$. For each $\omega \in \Sigma$, if we keep the count of the number of symbols in $\mathcal{T}$ smaller than or equal to ω in lexicographic order, then the desired symbol is simply the first symbol in lexicographic order whose count is $\geq j$. We can then continue taking steps forward from one neighbor to the next to recover additional symbols.

To encode Ψ, we conceptually partition Ψ into context lists according to each symbol $\omega \in \Sigma$. The context ω of $\Psi(i)$ is defined as the symbol $\mathcal{T}[SA[i]]$ that begins the suffix starting in position $SA[i]$. We denote a j-context as a length-j string. Suppose we have two suffixes at indices i and i' (where $i < i'$) in the suffix array that both start with the same symbol (context) ω. Then if we remove the first symbol ω from each suffix, the two suffixes that remain must maintain the same relative ordering in the suffix array:

▶ **Property 1.** *The neighbor function Ψ is a piecewise increasing sequence.*

We show the neighbor function Ψ for text $\mathcal{T} =$ `tcaaaatatatgcaacatatagtattagattgtat#` (see Table 1). The suffixes starting with the symbol `a` span indices $[1, 15]$ of the suffix array, and their Ψ values form an increasing sequence $\langle$2, 4, 5 ,11, 18, 19, 22, 23, 25, 27, 28, 29, 32, 34, 35$\rangle$, which we call `a`'s context list. For a given k (to be specified later), we further partition each ω's context list into sublists $\langle\omega, \mathsf{x}\rangle$ according to k-context $\mathsf{x} \in \Sigma^k$. Here k-context x of Ψ denotes the k-symbol prefix of the suffix starting at position $SA[\Psi(i)]$ and its preceding symbol is $SA[i]$. For the text $\mathcal{T}$ shown in Table 1, the symbol $SA[0] =$ `#` precedes the 3-context `tca` of the suffix starting at $SA[\Psi(0)] = SA[31]$. As another example, the 2-context `at` associated with indices $[8, 15]$ further decomposes `a`'s context list into the sublist $\langle 23, 25, 27, 28, 29, 32, 34, 35\rangle$. The entries in each sublist also form an increasing sequence.

The next property is key for efficient coding of the neighbor function Ψ:

▶ **Property 2.** *For any k-context $\mathsf{x} \in \Sigma^k$, all the entries in sublists $\langle\omega, \mathsf{x}\rangle$ for all nonempty contexts $\omega \in \Sigma$ form an index range of contiguous values in the suffix array SA. Moreover, all k-contexts x' that precede x in lexicographic order create an index range immediately prior to the index range of x in the suffix array.*

For coding purposes, this property allows us to normalize each entry $\Psi(i)$ in each sublist $\langle\omega, \mathsf{x}\rangle$ from 1 to $total(\mathsf{x})$, where $total(\mathsf{x})$ is the number of suffixes whose first $k + 1$ symbols are of the form $y\mathsf{x}$ for any $y \in \Sigma$. For example in Table 1, for $\mathsf{x} =$ `t`, the union of the sublists $\langle\omega, \mathtt{x}\rangle$ for all ω is the contiguous range $[23, 35]$ of 13 values.

In Section 4 we will see how the ability of Wavelet Trees to encode individual sublists with 0th-order entropy results in an cumulative higher-order compression for the full text $\mathcal{T}$.

3.2 FM-Index

The FM-index involves accessing the Burrows-Wheeler transform (BWT [1]), which appears as the L column in the example in Table 1. Adjacent entries in the L column tend to share similar statistics regarding which symbols follow because the symbols that follow them in the text are adjacent in lexicographic order. As an example, $L[8, 15]$ is the substring of L formed by all symbols of $\mathcal{T}$ that precede the 2-context `at`, as you can observe in Table 1.

The close relation between the CSA and BWT follows from the fact that the neighbor function Ψ is the inverse of the so-called LF function of the BWT (which maps the index of a symbol in the L column to the index where it appears when rotated forward in the F column). In other words, $\Psi(i) = j$ iff $LF(j) = i$, as can be seen from Table 1.

Building LF is straightforward for symbols that occur only once, as it is the case of `#` in our running example of Table 1. But computing LF efficiently is no longer trivial regarding recurring symbols. Nonetheless, it can be solved in optimal $\mathcal{O}(n)$ time thanks to a clever algorithm that uses an auxiliary vector C of size σ and is shown in Algorithm 1. For the

Algorithm 1 Constructing the LF-mapping from column L.

```
1: for $i = 0, 1, \ldots, n-1$ do
2:     $C[L[i]]$++;
3: $temp = 0,\ sum = 0$;
4: for $i = 0, 1, \ldots, \sigma$ do
5:     $temp = C[i]$;
6:     $C[i] = sum$;
7:     $sum$ += $temp$;
8: for $i = 0, 1, \ldots, n-1$ do
9:     $LF[i] = C[L[i]]$;
10:    $C[L[i]]$++;
```

sake of description, we assume that array C is indexed by a *symbol* rather than by an integer. The first for-loop in Algorithm 1 computes, for each symbol c, the number n_c of its occurrences in L, and thus it sets $C[c] = n_c$. Then, the second for-loop turns these symbol-wise occurrences into a cumulative sum, namely $C[c] = \sum_{x<c} n_x$. We notice that $C[c]$ gives the first position in F where the symbol c occurs. Therefore, before the last for-loop starts, $C[c]$ is the landing position in F of the first c in L (we thus know the LF-mapping for the first occurrence of every alphabet symbol). Finally, the last for-loop scans the column L, and whenever it encounters the symbol $L[i] = c$, it sets $LF[i] = C[c]$ and then increments $C[c]$. So the algorithm maintains the invariant that $LF[i] = \sum_{x<c} n_x + k$, after that k occurrences of c in L have been processed. The time complexity of such computation is $\mathcal{O}(n)$.

Like the neighbor function Ψ in the CSA, the BWT transform suffices to recreate the original text $\mathcal{T}$. To do so, it uses a backward scan supported by the LF-mapping in $\mathcal{O}(n)$ time and space. The algorithm starts from the last symbol of $\mathcal{T}$, which occurs at $L[0]$; and then it proceeds by moving one symbol at a time to the left in $\mathcal{T}$, deploying the properties of the LF-mapping: if we know the symbol in the F column at index i, then the preceding symbol in the text is the symbol in the L column on the same row i. Given this observation, the backward reconstruction of $\mathcal{T}$ first maps the current symbol occurring in L, say $L[i]$, to its corresponding copy in F, hence $j = LF[i]$; and then takes the symbol $L[j]$ that is ensured by the properties of the cyclic rotations of the rows of the BWT to precede $F[j] = L[i]$ in $\mathcal{T}$ (see above). This double step, which returns the algorithmic focus on L, allows to move one symbol leftward in $\mathcal{T}$. Hence, repeating this for n steps, we can reconstruct the original input string $\mathcal{T}$ in $\mathcal{O}(n)$ time. As far as the construction of the BWT is concerned, we mention that it can be done by building the suffix array data structure via a plethora of time and space-efficient algorithms that now abound in the literature (see e.g. [17, 16, 22]) or directly via several elegant approaches (see e.g. [14, 25, 18, 23]).

But how useful is it to permute $\mathcal{T}$ via the BW-Transform? If we read the first column F of Table 1, we notice a sequence of possibly long runs of equal symbols. It is clear that by compressing that sequence with a simple RLE-compressor (namely, the one that substitutes each run of equal symbols x^h with a pair $\langle x, h\rangle$) we would get a high compression ratio. But F does not allow us to return to the original string $\mathcal{T}$ because all strings consisting of 15 `a`, 3 `c`, 4 `g`, and 13 `t` symbols would obtain the same F. It can be proved [1] that the only column allowing to return to $\mathcal{T}$, for all possible $\mathcal{T}$, is properly L. That column could be highly compressible because it satisfies the so-called locally homogeneous property, namely, that substrings of L consist of a few distinct symbols. The reason comes from the fact that if we take a string (context) of length k (as in the Definition 3 in `PART I` of kth-order entropy), say $\mathsf{x} \in \Sigma^k$, the n_{x} symbols $\mathcal{T}_{\mathsf{x}}$ immediately preceding each occurrence of x in $\mathcal{T}$ occur contiguously in L. The nice issue here is that many real sources (they are called Markovian) do exist that generate data sequences, other than texts, that can be turned to be locally homogeneous via the Burrows-Wheeler Transform and thus can be highly compressed by simpler compressors.

This basic principle is deployed by the design of the well-known `bzip` tool (now[1] arrived to version 3), whose algorithmic pipeline we briefly sketch: Let us for a moment focus on two simple algorithms, namely, the *Move-To-Front* (shortly, MTF) and the *Run-Length Encoding* (shortly RLE, mentioned above). The former maps symbols into integers, and the latter maps runs of equal symbols into pairs. We observe that RLE is a compressor indeed, because the output sequence may be reduced in length in the presence of long runs of equal symbols; while MTF is a transform that can be turned into a compressor by encoding the integers via proper variable-length encoders. In general, the compression performance of those algorithms is very poor; while BWT turns them magically to be very effective if combined together: first apply MTF on L, then input the result on RLE, and finally encode its output with a statistical coder like Huffman or arithmetic coding. That's it[2].

Manzini [21] showed that this, and other pipelines based upon the BWT, can represent a text string $\mathcal{T}$ of n symbols using space proportional to n times its kth-order empirical entropy $\mathcal{H}_k(\mathcal{T})$, with guarantees stronger than the ones ensured by Lempel-Ziv-based compressors (such as gzip). Later analysis using Wavelet Trees reduced the constant factor in front of the $n\mathcal{H}_k(\mathcal{T})$ term to 1 [11, 8]. Manzini's result generated increased interest in the BWT transform, whose impact on data compression, and not just compressed text indexing, was highlighted by [5] as a sort of *compression booster*.

4 Compression and Boosting to Higher-Order Entropy

The original motivation for Wavelet Trees was to extend the RRR method [26] from binary sequences to sequences composed of symbols from general alphabets in order to achieve 0th-order entropy. That capability is important because if a text $\mathcal{T}$'s Ψ neighbor function values [12, 11, 27] (in the case of compressed suffix arrays) and the BWT transform (L column) [7, 24] (in the case of the FM-index) are decomposed into substrings based upon $\mathcal{T}$'s contexts of length k, then we can use a Wavelet Tree to compress each of those substrings to its 0th-order entropy (plus lower-order terms). The key point is that the resulting cumulative encoding will achieve a compression equal to the kth-order entropy of $\mathcal{T}$ (plus lower-order terms).

This notion of achieving a high-order entropy based upon individual encodings of 0th-order entropy (later called "boosting" in [3, 5]) is inherent in Definition 3 in `PART I` of kth-order entropy, in which $n\mathcal{H}_k(\mathcal{T})$ is expressed as a sum over all k-contexts x of the terms $n_{\mathsf{x}}\mathcal{H}_0(\mathcal{T}_{\mathsf{x}})$. Grossi, Gupta, and Vitter used Wavelet Trees to provide the first demonstration of the asymptotically optimal space bound $\sim n\mathcal{H}_k(\mathcal{T})$ (i.e., with leading coefficient 1) for compressed suffix arrays and for the Burrows-Wheeler transform [11].

In order to better illustrate this powerful idea, let us consider a generic 0-order statistical compressor $\mathcal{C}_0$ whose performance, in bits per symbol, over a string s is bounded by $\mathcal{H}_0(s) + f(|s|)$. We notice that the function $f(|s|) = 2/(|s|)$ is the one achieved by arithmetic coding and it is $f(|s|) = 1$ for Huffman coding. In order to turn $\mathcal{C}_0$ into an effective kth-order compressor $\mathcal{C}_k$, for a fixed $k > 0$, we can compute the Burrows-Wheeler Transform of the

[1] See `https://en.wikipedia.org/wiki/Bzip2` and `https://github.com/kspalaiologos/bzip3`
[2] For a scholarly discussion of the BWT and the FM-index, the reader is directed to the book [2].

input string $\mathcal{T}$, take all possible length-k substrings x of $\mathcal{T}$, and then partition the column L into the substrings $\mathcal{T}_{\mathsf{x}}$ (each one formed by the last symbols of the rows prefixed by x). Therefore, the final step is just to compress each $\mathcal{T}_{\mathsf{x}}$ by $\mathcal{C}_0$, and concatenate the output bit sequences by alphabetically increasing x.

As far as the compression performance in bits per symbol is concerned, we easily derive that it can be bounded as $(1/n)\sum_{\mathsf{x}\in\Sigma^k} |\mathcal{T}_{\mathsf{x}}|\ (\mathcal{H}_0(\mathcal{T}_{\mathsf{x}}) + f(|\mathcal{T}_{\mathsf{x}}|)) = \mathcal{H}_k(\mathcal{T}) + \sum_{\mathsf{x}\in\Sigma^k} |\mathcal{T}_{\mathsf{x}}|\ (f(|\mathcal{T}_{\mathsf{x}}|)/n)$ bits, where we have applied Definition 3 in `PART I` of $\mathcal{H}_k(\mathcal{T})$ to the summation of the $\mathcal{H}_0(\mathcal{T}_x)$. Now, if $\mathcal{C}_0$ is the arithmetic coder, the previous bound is $\mathcal{H}_k(\mathcal{T}) + \mathcal{O}(\sigma^k/n)$ bits, since $f(|s|) = 2/|s|$. In [5] the authors showed other stronger upper bounds to the performance of $\mathcal{C}_k$ and, more importantly, that one actually does not need to fix k in advance, since there is a so-called Compression Booster that identifies in optimal linear time a partition of L that achieves a compression ratio that is at least as good as the one obtained by $\mathcal{C}_k$, up to an additive lower-order term for any possible k in the allowable range. The algorithm is a surprisingly simple and elegant greedy-based algorithm; we refer the interested reader to that paper.

4.1 Using a Single Wavelet Tree with Implicit Partitions of Context

In the previous section, we decomposed the string being compressed (such as the neighbour sequence Ψ of the CSA or the L column in the FM Index) into contexts. Each individual context $\mathcal{T}_{\mathsf{x}}$ was then encoded to its 0th-order entropy, and by Definition 3 in `PART I` of kth-order entropy, the sum of the resulting individual $n_{\mathsf{x}}\,\mathcal{H}_0(\mathcal{T}_{\mathsf{x}})$ terms resulted in a total of $n\,\mathcal{H}_k(\mathcal{T})$ bits (the boosting effect). In some cases, the results hold for all k in the allowed range $k < \alpha \log_\sigma n$ for some constant $0 < \alpha < 1$.

Foschini et al. [9] showed how order kth-order entropy can be achieved by using a single Wavelet Tree without explicit partitioning into k-contexts. Intuitively, there are two reasons:

- One reason is that the individual Wavelet Trees that would be constructed for each context can be found within the single Wavelet Tree constructed on the entire BWT string L, as illustrated in Figure 1.
- The other reason follows from using a method like run-length encoding (RLE) or gap encoding (Gap) (in conjunction with a prefix coder such as the γ or δ code to encode the lengths) in order to encode the bit arrays of the nodes of the single Wavelet Tree. On the plus side, there is thus no need for data structures to keep track of the statistics of many individual contexts. On the negative side, the coding method RLE or Gap does not know the k-context boundaries. However, Foschini et al. [9] showed that RLE can adapt to the statistics for each k-context it encounters, and the positives make up for the negatives. By superimposing hypothetical k-context boundaries on the string for purposes of analysis, the resulting RLE implicitly encodes each k-context in roughly twice 0th-order space, and by Definition 3 in `PART I` of kth-order entropy, the resulting overall encoding achieves $2n\,\mathcal{H}_k(\mathcal{T}_k)$ bits leading space term. In effect, it achieves implicit boosting.

In implicit boosting, there is no need to specify k in advance; the compression results hold for all k in the allowed range $k < \alpha \log_\sigma n,\ \ 0 < \alpha < 1$. Foschini et al. [9] used RLE encoding with γ encoding, but other coding methods could be used, possibly switching dynamically among several methods (along with the bits that indicate the choices made) [3, 13, 15].

Gog et al. [10] compress the BWT by using the RRR method [26] to encode the Wavelet Trees of fixed-size blocks of the BWT rather than using partitions based upon higher-order contexts; they showed that the resulting compression achieves the $n\,\mathcal{H}_k(\mathcal{T})$ leading space term. Mäkinnen and Navarro [19, 20] showed the surprising result that RRR [26] applied

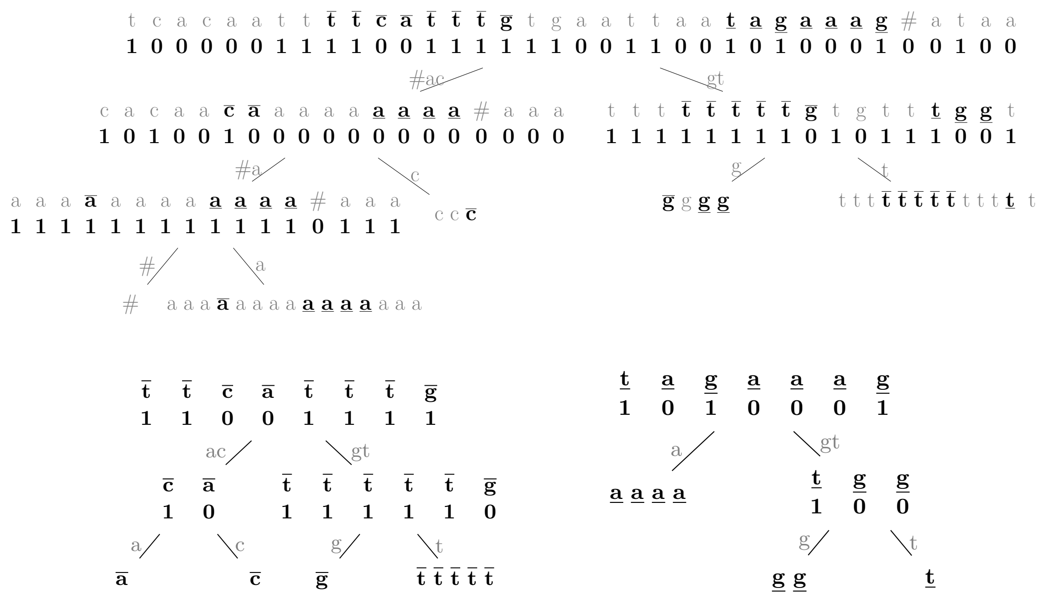

Figure 1 The top row shows a single Wavelet Tree for the BWT L of $\mathcal{T}$ as in Table 1, and the bottom row shows separate Wavelet Trees for each of the context blocks `ttcatttg`, highlighted as overlined letters, and `tagaaag`, highlighted as underlined letters, of L corresponding to 2-contexts `at` and `ta`, respectively.

to a single Wavelet Tree of the entire BWT sequence or Ψ sequence (i.e., without any partitioning) also achieves the $n\,\mathcal{H}_k(\mathcal{T}_k)$ bits leading space term, for a similar reason as in [9] described above; careful analysis in [19, 20] showed that the sublocks formed by [26] within each Wavelet Tree node implicitly encode each kth-order context in 0th-order entropy space plus lower-order terms. Grossi, Vitter, and Xu [13] present interesting experiments on the practical performance of these approaches.

References

1 M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm, 1994.
2 Paolo Ferragina. *Pearls of Algorithm Engineering*. Cambridge University Press, 2023.
3 Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. The myriad virtues of Wavelet Trees. *Information and Computation*, 207(8):849–866, 2009. `doi:10.1016/j.ic.2008.12.010`.
4 Paolo Ferragina, Raffaele Giancarlo, Giovanni Manzini, Giovanna Rosone, Rossano Venturini, and Jeffrey Scott Vitter. Wavelet Tree, Part I: A Brief History. appearing in the Festschrift dedicated to Roberto Grossi.
5 Paolo Ferragina, Raffaele Giancarlo, Giovanni Manzini, and Marinella Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, 52(4):688–713, July 2005. `doi:10.1145/1082036.1082043`.
6 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science (FOCS'00)*, pages 390–398, Washington, DC, November 2000. IEEE Computer Society. `doi:10.1109/SFCS.2000.892127`.
7 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005. `doi:10.1145/1082036.1082039`.
8 Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):Article 20, 2007. An earlier version appeared in *Proceedings of the 11th Symposium on String Processing and Information Retrieval (SPIRE)*, 150–160, October 2004.

9 Luca Foschini, Roberto Grossi, Ankur Gupta, and Jeffrey S. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. *ACM Transactions on Algorithms*, 2(4):611–639, 2006. This work combined earlier versions from *SIAM Symposium on Discrete Algorithms (SODA)*, January 2004, and "Fast compression with a static model in high-order entropy," *Data Compression Conference (DCC)*, March 2004. `doi:10.1145/1198513.1198521`.

10 Simon Gog, Juha Kärkkäinen, Dominik Kempa, Matthias Petri, and Simon J. Puglisi. Fixed block compression boosting in FM-Indexes: Theory and practice. *Algorithmica*, 81:1370–1391, 2019. `doi:10.1007/S00453-018-0475-9`.

11 Roberto Grossi, Ankur Gupta, and Jeffrey S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms (SODA'03)*, pages 841–850, New York, NY, January 2003. Association for Computing Machinery.

12 Roberto Grossi and Jeffrey S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005. An earlier version appeared in *Proceedings of the 32nd ACM Symposium on Theory of Computing (STOC'00)*, May 2000, 397–406. `doi:10.1137/S0097539702402354`.

13 Roberto Grossi, Jeffrey S. Vitter, and Bojian Xu. Wavelet trees: From theory to practice. In *2011 First International Conference on Data Compression, Communications and Processing*, pages 210–221, June 2011. `doi:10.1109/CCP.2011.16`.

14 Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Breaking a time-and-space barrier in constructing full-text indices. *SIAM J. Comput.*, 38(6):2162–2178, 2009. `doi:10.1137/070685373`.

15 Hongwei Huo, Zongtao He, Pengfei Liu, and Jeffrey S. Vitter. FM-Adaptive: A practical data-aware FM-index, 2025. submitted.

16 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006. `doi:10.1145/1217856.1217858`.

17 Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Constructing suffix arrays in linear time. *Journal of Discrete Algorithms*, 3(2):126–142, 2005. `doi:10.1016/J.JDA.2004.08.019`.

18 Alan Kuhnle, Taher Mun, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini. Efficient construction of a complete index for pan-genomics read alignment. In Lenore J. Cowen, editor, *Research in Computational Molecular Biology - 23rd Annual International Conference, RECOMB 2019, Washington, DC, USA, May 5-8, 2019, Proceedings*, volume 11467 of *Lecture Notes in Computer Science*, pages 158–173. Springer, 2019. `doi:10.1007/978-3-030-17083-7_10`.

19 Veli Mäkinen and Gonzalo Navarro. Implicit compression boosting with applications to self-indexing. In *Proceedings of the 14th International Symposium on String Processing and Information Retrieval (SPIRE'07)*, pages 229–241, Heidelberg, Germany, 2007. Springer-Verlag Berlin. `doi:10.1007/978-3-540-75530-2_21`.

20 Veli Mäkinen and Gonzalo Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms*, 4(3):article 32, 2008.

21 Giovanni Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001. An earlier version appeared in *Proceedings of the 10th Symposium on Discrete Algorithms (SODA '99)*, January 1999, 669–677. `doi:10.1145/382780.382782`.

22 Giovanni Manzini and Paolo Ferragina. Engineering a lightweight suffix array construction algorithm. In Rolf H. Möhring and Rajeev Raman, editors, *Algorithms - ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17-21, 2002, Proceedings*, volume 2461 of *Lecture Notes in Computer Science*, pages 698–710. Springer, 2002. `doi:10.1007/3-540-45749-6_61`.

23 J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. Fast compressed self-indexes with deterministic linear-time construction. *Algorithmica*, 82(2):316–337, 2020. A conference version of this paper appeared in Proc. ISAAC 2017. `doi:10.1007/S00453-019-00637-X`.

24 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):Article 2, 2007.

25 Daisuke Okanohara and Kunihiko Sadakane. A linear-time burrows-wheeler transform using induced sorting. In Jussi Karlgren, Jorma Tarhio, and Heikki Hyyrö, editors, *String Processing and Information Retrieval, 16th International Symposium, SPIRE 2009, Saariselkä, Finland, August 25-27, 2009, Proceedings*, volume 5721 of *Lecture Notes in Computer Science*, pages 90–101. Springer, 2009. doi:10.1007/978-3-642-03784-9_9.

26 Rajeev Raman, Venkatesh Raman, and S.Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):Article 43, 2007. doi:10.1145/1290672.1290680.

27 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007. doi:10.1007/S00224-006-1198-X.

28 Kunihiko Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proceedings of the 11th Symposium on Algorithms and Computation (ISAAC'00)*, pages 410–421, Banff Alberta, Canada, August 2000. Springer-Verlag Berlin, Heidelberg.