Wavelet Trees for All [⋆]

Gonzalo Navarro

Dept. of Computer Science, University of Chile. gnavarro@dcc.uchile.cl

Abstract. The wavelet tree is a versatile data structure that serves a number of purposes, from string processing to geometry. It can be regarded as a device that represents a sequence, a reordering, or a grid of points. In addition, its space adapts to various entropy measures of the data it encodes, enabling compressed representations. New competitive solutions to a number of problems, based on wavelet trees, are appearing every year. In this survey we give an overview of wavelet trees and the surprising number of applications in which we have found them useful: basic and weighted point grids, sets of rectangles, strings, permutations, binary relations, graphs, inverted indexes, document retrieval indexes, full-text indexes, XML indexes, and general numeric sequences.

1 Introduction

The *wavelet tree* was invented in 2003 by Grossi, Gupta, and Vitter [54], as a data structure to represent a sequence and answer some queries on it. Curiously, a data structure that has turned out to have a myriad of applications was buried in a paper full of other eye-catching results. The first mention to the name "wavelet tree" appears on page 8 of 10 [54, Sec. 4.2]. The last mention is also on page 8, save for a figure caption on page 9. Yet, the wavelet tree was a key tool to obtain the main result of the paper, a milestone in compressed full-text indexing.

It is interesting that, after some thought, one can see that the wavelet tree is a slight generalization of an old (1988) data structure by Chazelle [25], heavily used in Computational Geometry. This data structure represents a set of points on a two-dimensional grid: it describes a successive reshuffling process where the points start sorted by one coordinate and end up sorted by the other. Kärkkäinen, in 1999 [66], was the first to put this structure in use in the completely different context of text indexing. Still, the concept and usage were totally different from the one Grossi et al. would propose four years later.

We have already mentioned three ways in which wavelet trees can be regarded: (*i*) as a representation of a sequence; (*ii*) as a representation of a reordering of elements; (*iii*) as a representation of a grid of points. Since 2003, these views of wavelet trees, and their interactions, have been fruitful in a surprisingly wide range of problems, extending well beyond the areas of text indexing and computational geometry where the structure was conceived.

[⋆] Partially funded by Millennium Nucleus Information and Coordination in Networks ICM/FIC P10-024F, Chile.

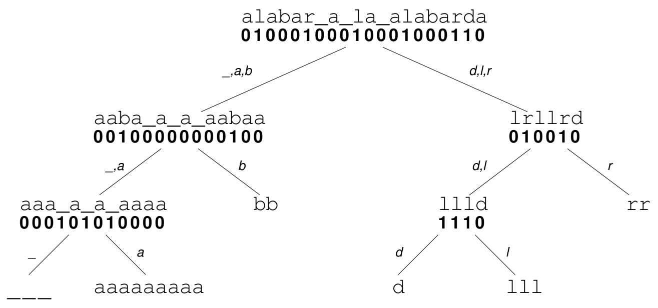

Fig. 1. A wavelet tree on string $S =$ "`alabar a la alabarda`". We draw the spaces as underscores. The subsequences of S and the subsets of Σ labeling the edges are drawn for illustration purposes; the tree stores only the topology and the bitmaps.

Our goal in this article is to give an overview of this marvellous data structure and its many applications. We aim to introduce, to an audience with a general algorithmic background, the basic data organization used by wavelet trees, the information they can model, and the wide range of problems they can solve. We will also mention the most technical results and give the references to be followed by the more knowledgeable readers, advising the rest what to skip.

Being ourselves big fans of wavelet trees, and having squeezed them out for several years, it is inevitable that there will be many references to our own work in this survey. We apologize in advance for this, as well as for oversights of others' results, which are likely to occur despite our efforts.

2 Data Structure

Let $S[1,n] = s_1 s_2 \ldots s_n$ be a sequence of symbols $s_i \in \Sigma$, where $\Sigma = [1..\sigma]$ is called the *alphabet*. Then S can be represented in plain form using $n\lceil \lg \sigma \rceil = n \lg \sigma + O(n)$ bits (we use $\lg x = \log_2 x$).

Structure. A wavelet tree [54] for sequence $S[1,n]$ over alphabet $[1..\sigma]$ can be described recursively, over a sub-alphabet range $[a..b] \subseteq [1..\sigma]$. A wavelet tree over alphabet $[a..b]$ is a binary balanced tree with $b - a + 1$ leaves. If $a = b$, the tree is just a leaf labeled a. Else it has an internal root node, v_{root}, that represents $S[1,n]$. This root stores a bitmap $B_{v_{root}}[1,n]$ defined as follows: if $S[i] \leq (a+b)/2$ then $B_{v_{root}}[i] = 0$, else $B_{v_{root}}[i] = 1$. We define $S_0[1,n_0]$ as the subsequence of $S[1,n]$ formed by the symbols $c \leq (a+b)/2$, and $S_1[1,n_1]$ as the subsequence of $S[1,n]$ formed by the symbols $c > (a+b)/2$. Then, the left child of v_{root} is a wavelet tree for $S_0[1,n_0]$ over alphabet $[a..\lfloor (a+b)/2 \rfloor]$ and the right child of v_{root} is a wavelet tree for $S_1[1,n_1]$ over alphabet $[1 + \lfloor (a+b)/2 \rfloor..b]$.

Fig. 1 displays a wavelet tree for the sequence $S =$ "`alabar a la alabarda`". Here for legibility we are using $\Sigma = \{$`' '`, `a`, `b`, `d`, `l`, `r`$\}$, so $n = 19$ and $\sigma = 6$.

Note that this wavelet tree has height $\lceil \lg \sigma \rceil$, and it has σ leaves and $\sigma - 1$ internal nodes. If we regard it level by level, it is not hard to see that it stores exactly n bits at each level, and at most n bits in the last one. Thus, $n\lceil\lg\sigma\rceil$ is an upper bound to the total number of bits it stores. Storing the topology of the tree requires $O(\sigma\lg n)$ further bits, if we are careful enough to use $O(\lg n)$ bits for the pointers. This extra space may be a problem on large alphabets. We show in the paragraph "Removing redundancy" how to save it.

Tracking symbols. This wavelet tree represents S, in the sense that one can recover S from it. More than that, it is a *succinct data structure* for S, in the sense that it takes space asymptotically equal to a plain representation of S, and it permits accessing any $S[i]$ in time $O(\lg\sigma)$, as follows.

To extract $S[i]$, we first examine $B_{v_{root}}[i]$. If it is a 0, we know that $S[i] \leq (\sigma+1)/2$, otherwise $S[i] > (\sigma+1)/2$. In the first case, we must continue recursively on the left child; in the second case, on the right child. The problem is to determine where has position i been mapped to on the left (or right) child. In the case of the left child, where $B_{v_{root}}[i] = 0$, i has been mapped to position i_0, which is the number of 0s in $B_{v_{root}}$ up to position i. For the right child, where $B_{v_{root}}[i] = 1$, this corresponds to position i_1, the number of 1s in $B_{v_{root}}$ up to position i. The number of 0s (resp. 1s) up to position i in a bitmap B is called $\mathtt{rank}_0(B,i)$ (resp. $\mathtt{rank}_1(B,i)$). We continue this process recursively until we arrive at a leaf. The label of this leaf is $S[i]$. Note that we do not store the leaf labels; those are deduced as we successively restrict the subrange $[a..b]$ of $[1..\sigma]$ as we descend.

Operation `rank` was already considered by Chazelle [25], who gave a simple data structure using $O(n)$ bits for a bitmap $B[1,n]$, that computed `rank` in constant time (note that we only have to solve $\mathtt{rank}_1(B,i)$, since $\mathtt{rank}_0(B,i) = i - \mathtt{rank}_1(B,i)$). Jacobson [63] improved the space to $n + O(n\lg\lg n/\lg n) = n + o(n)$ bits, and Golynski [48, 49] proved this space is optimal as long as we maintain B in plain form and build extra data structures on it. The solution is, essentially, storing `rank` answers every $s = \lg^2 n$ bits of B (using $\lg n$ bits per sample), then storing `rank` answers relative to the last sample every $(\lg n)/2$ bits (using $\lg s = 2\lg\lg n$ bits per sub-sample), and using a universal table to complete the answer to a `rank` query within a sub-sample. We will use in this survey the notation $\mathtt{rank}_b(B,i,j) = \mathtt{rank}_b(B,j) - \mathtt{rank}_b(B,i-1)$.

Above, we have *tracked* a position from the root to a leaf, and as a consequence we have discovered the symbol represented at the root position. It is also useful to carry out the inverse process: given a position at a leaf, we can track it upwards and find out where it is on the root bitmap. This is done as follows.

Assume we start at a given leaf, at position i. If the leaf is the left child of its parent v, then the position i' corresponding to i at v is the i-th occurrence of a 0 in its bitmap B_v. If the leaf is the right child of its parent v, then i' is the position of the i-th occurrence of a 1 in B_v. This procedure is repeated from v until we reach the root, where we find the final position. The operation of finding the i-th 0 (resp. 1) in a bitmap $B[1,n]$ is called $\mathtt{select}_0(B,i)$ (resp. $\mathtt{select}_1(B,i)$), and it can also be solved in constant time using the n bits of B plus $o(n)$ bits [27, 79]. Thus the time to track a position upwards is also $O(\lg\sigma)$.

The constant-time solution for `select` [27, 79] is analogous to that of `rank`. The bitmap is cut into blocks with s 1s. Those that are long enough to store all their answers within sublinear space are handled in this way. The others are not too long (i.e., $O(\lg^{O(1)} n)$) and thus encoding positions inside them require fewer bits (i.e., $O(\lg \lg n)$). This permits repeating the idea recursively a second time. The third time, the remaining blocks are so short that can be handled in constant time using universal tables. Golynski [48, 49] reduced the $o(n)$ extra space to $O(n \lg \lg n / \lg n)$ and proved this is optimal if B is stored in plain form.

With the support for `rank` and `select`, the space required by the basic binary balanced wavelet tree reaches $n\lceil \lg \sigma \rceil + o(n) \lg \sigma + O(\sigma \lg n)$ bits. This completes a basic description of wavelet trees; the rest of the section is more technical.

Reducing redundancy. As mentioned, the $O(\sigma \lg n)$ term can be removed if necessary [72, 74]. We slightly alter the balanced wavelet tree shape, so that all the leaves are grouped to the left (for this sake we divide the interval $[a..b]$ of $[1..\sigma]$ into $[a..a + 2^{\lceil \lg(b-a+1) \rceil -1} - 1]$ and $[a + 2^{\lceil \lg(b-a+1) \rceil -1}..b]$). Then, all the bitmaps at all the levels belong to consecutive nodes, and they can all be concatenated into a large bitmap $B[1, n\lceil \lg \sigma \rceil]$. We know the bitmap of level ℓ starts at position $1 + n(\ell - 1)$. Moreover, if we have determined that the bitmap of a wavelet tree node corresponds to $B[l, r]$, then the bitmap of its left child is at $B[n + l, n + l + \mathtt{rank}_0(B, l, r) - 1]$, and that of the right child is at $B[n + l + \mathtt{rank}_0(B, l, r), n + r]$. Moving to the parent of a node is more complicated, but upward traversals can always be handled by first going down from the root to the desired leaf, so as to discover all the ranges in B of the nodes in the path, and then doing the upward processing as one returns from the recursion.

Using just one bitmap, we do not need pointers for the topology, and the overall space becomes $n\lceil \lg \sigma \rceil + o(n) \lg \sigma$ bits. The time complexities do not change (albeit in practice the operations are slowed down a bit due to the extra `rank` operations needed to navigate [28]).

The redundancy can be further reduced by representing the bitmaps using a structure by Golynski et al. [50], which uses $n + O(n \lg \lg n / \lg^2 n)$ bits and supports constant-time `rank` and `select` (this representation does not leave the bitmap in plain form, and thus it can break the lower bound [49]). Added over all the wavelet tree bitmaps, the space becomes $n \lg \sigma + O(n \lg \sigma \lg \lg n / \lg^2 n) = n \lg \sigma + o(n)$ bits.[1] This structure has not been implemented as far as we know.

Speeding up traversals. Increasing the arity of wavelet trees reduces their height, which dictates the complexity of the downward and upward traversals.

[1] We assume $\lg \sigma = O(\lg n)$ here; otherwise there are many symbols that do not appear in S. If this turns out to be the case, one should use a mapping from Σ to the range $[1..\sigma']$, where $\sigma' \leq n$ is the number of symbols actually appearing in S. Such a mapping takes constant time and $\sigma' \lg(\sigma/\sigma') + o(\sigma') + O(\lg \lg \sigma)$ bits of space using the "indexable dictionaries" of Raman et al. [93]. Added to the $n \lg \sigma' + o(n)$ bits of the wavelet tree, we are within $n \lg \sigma + o(n) + O(\lg \lg \sigma)$ bits. This is $n \lg \sigma + o(n)$ unless $n = O(\lg \lg \sigma)$, in which case a plain representation of S using $n\lceil \lg \sigma \rceil$ bits solves all the operations in $O(\lg \lg \sigma)$ time. To simplify, a recent analysis [45] claims $n \lg \sigma + O(n)$ bits under similar assumptions. We will ignore the issue from now, and assume for simplicity that all symbols in $[1..\sigma]$ do appear in S.

If the wavelet tree is d-ary, then its height is $\lceil \lg_d \sigma \rceil$. However, the wavelet tree does not store bitmaps anymore, but rather sequences B_v over alphabet $[1..d]$, so that the symbol at $S_v[i]$ is stored at the child numbered $B_v[i]$ of node v.

In order to obtain time complexities $O(1 + \lg_d \sigma)$ for the operations, we need to handle `rank` and `select` on sequences over alphabet $[1..d]$, in constant time. Ferragina et al. [40] showed that this is indeed possible, while maintaining the overall space within $n \lg \sigma + o(n) \lg \sigma$, for $d = o(\lg n / \lg \lg n)$. Using, for example, $d = \lg^{1-\epsilon} n$ for any constant $0 < \epsilon < 1$, the overall space is $n \lg \sigma + O(n \lg \sigma / \lg^{\epsilon} n)$ bits. Golynski et al. [50] reduced the space to $n \lg \sigma + o(n)$ bits.

To support symbol `rank` and `select` on a sequence $R[1, n]$ over alphabet $[1..d]$, we assume we have d bitmaps $B_c[1, n]$, for $c \in [1..d]$, where $B_c[i] = 1$ iff $R[i] = c$. Then $\mathtt{rank}_c(R, i)$ and $\mathtt{select}_c(R, i)$ are reduced to $\mathtt{rank}_1(B_c, i)$ and $\mathtt{select}_1(B_c, i)$. We cannot afford to store those B_c, but we can store their extra $o(n)$ data for binary `rank` and `select`. Each time we need access to B_c, we access instead R and use a universal table to simulate the bitmap's content. Such table gives constant-time access to chunks of length $\lg_d(n)/2$ instead of $\lg(n)/2$, so the overall space using Golynski et al.'s bitmap index representation [48, 49] is $O(dn \lg \lg n / \lg_d n)$, which added over the $\lg_d \sigma$ levels of the wavelet tree gives $O(n \lg \sigma \cdot d \lg d \lg \lg n / \lg n)$. This is $o(n \lg \sigma)$ for any $d = \lg^{1-\epsilon} n$. Further reducing the redundancy to $o(n)$ bits requires more sophisticated techniques [50].

Thus, the $O(\lg \sigma)$ upward/downward traversal times become $O(\lg \sigma / \lg \lg n)$ with multiary wavelet trees. Although theoretically attractive, it is not easy to translate their advantages to practice (see, e.g., a recent work studying interesting practical alternatives [17]). An exception, for a particular application, is described in the paragraph "Positional inverted indexes" of Section 5).

The upward traversal can be speeded up further, using techniques known in computational geometry [25]. Imagine we are at a leaf u representing a sequence $S[1, n_u]$ and want to directly track position i to an ancestor v at distance t, which represents sequence $S[1, n_v]$. We can store at the leaf u a bitmap $B_u[1, n_v]$, so that the n_u positions corresponding to leaf u are marked as 1s in B_u. This bitmap is sparse, so it is stored in compressed form as an "indexable dictionary" [93], which uses $n_u \lg(n_v/n_u) + o(n_u) + O(\lg \lg n_v)$ bits and can answer $\mathtt{select}_1(B_u, i)$ queries in $O(1)$ time. Thus we track position i upwards for t levels in $O(1)$ time.

The space required for all the bitmaps that point to node v is the sum, over at most 2^t leaves u, of those $n_u \lg(n_v/n_u) + o(n_u) + O(\lg \lg n_v)$ bits. This is maximized when $n_u = n_v/2^t$ for all those u, where the space becomes $t \cdot n_v + o(n_v) + O(2^t \lg \lg n_v)$. Added over all the wavelet tree nodes with height multiple of t, we get $n \lg \sigma + o(n \lg \sigma) + O(\sigma \lg \lg n) = n \lg \sigma + o(n \lg \sigma)$. This is in addition to those $n \lg \sigma + o(n)$ bits already used by the wavelet tree.

If we want to track only from the leaves to the root, we may just use $t = \lg \sigma$ and do the tracking in constant time. In many cases, however, one wishes to track from arbitrary to arbitrary nodes. In this case we can use $1/\epsilon$ values of $t = \lg^{i\epsilon} \sigma$, for $i \in [1..1/\epsilon - 1]$, so as to carry out $O(\lg^{\epsilon} \sigma)$ upward steps with one value of t before reaching the next one. This gives a total complexity for upward traversals of $O((1/\epsilon) \lg^{\epsilon} \sigma)$ using $O((1/\epsilon) n \lg \sigma)$ bits of space.

Construction. It is easy to build a wavelet tree in $O(n \lg \sigma)$ time, by a linear-time processing at each node. It is less obvious how to do it in little extra space, which may be important for succinct data structures. Two recent results [31, 96] offer various relevant space-time tradeoffs, building the wavelet tree within the time given, or close, and asymptotically negligible extra space.

3 Compression

The wavelet tree adapts elegantly to the compressibility of the data in many ways. Two key techniques to achieve this are using specific encodings on bitmaps, and altering the tree shape. This whole section is technical, yet nonexpert readers may find inspiring the beginning of the paragraph "Entropy coding", and the paragraph "Changing shape".

Entropy coding. Consider again Fig. 1. The fact that the `'a'` is much more frequent than the other symbols translates into unbalanced 0/1 frequencies in various bitmaps. Dissimilarities in symbol frequencies are an important source of compressibility. The amount of compression that can be reached is measured by the so-called *empirical zero-order entropy* of a sequence $S[1, n]$:

$$H_0(S) \quad = \quad \sum_{c \in \Sigma} (n_c/n) \lg(n/n_c) \quad \leq \quad \lg \sigma$$

where n_c is the number of occurrences of c in S and the sum considers only the symbols that do appear in S. Then $nH_0(S)$ is the least number of bits into which S can be compressed by always encoding the same symbol in the same way.[2]

Grossi et al. [54] already showed that, if the bitmaps of the wavelet tree are compressed to their zero-order entropy, then their overall space is $nH_0(S)$. Let $B_{v_{root}}$ contain n_0 0s and n_1 1s. Then zero-order compressing it yields space $n_0 \lg(n/n_0) + n_1 \lg(n/n_1)$. Now consider its left child v_l. Its bitmap, B_{v_l}, is of length n_0, and say it contains n_{00} 0s and n_{01} 1s. Similarly, the right child is of length n_1 and contains n_{10} 0s and n_{11} 1s. Adding up the zero-order compressed space of both children yields $n_{00} \lg(n_0/n_{00}) + n_{01} \lg(n_0/n_{01}) + n_{10} \lg(n_1/n_{10}) + n_{11} \lg(n_1/n_{11})$. Now adding the space of the root bitmap yields $n_{00} \lg(n/n_{00}) + n_{01} \lg(n/n_{01}) + n_{10} \lg(n/n_{10}) + n_{11} \lg(n/n_{11})$. This would already be $nH_0(S)$ if $\sigma = 4$. It is easy to see that, by splitting the spaces of the internal nodes until reaching the wavelet tree leaves, we arrive at $\sum_{c \in \Sigma} n_c \lg(n/n_c) = nH_0(S)$.

This enables using any zero-order entropy coding for the bitmaps that supports constant-time `rank` and `select`. One is the "fully-indexable dictionary" of Raman et al. [93], which for a bitmap $B[1, n]$ requires $nH_0(B) + O(n \lg \lg n / \lg n)$ bits. A theoretically better one is that of Golynski et al. [50], which we have already mentioned without yet telling that it actually compresses the bitmap, to $nH_0(B) + O(n \lg \lg n / \lg^2 n)$. Pătraşcu [91] showed this can be squeezed up to

[2] In classical information theory [32], H_0 is the least number of bits per symbol achievable by any compressor on an infinite source that emits symbols independently and randomly with probabilities n_c/n.

$nH_0(B)+O(n/\lg^c n)$, answering `rank` and `select` in time $O(c)$, for any constant c, and that this is essentially optimal [92].

Using the second or third encoding, the wavelet tree represents S within $nH_0(S)+o(n)$ bits, still supporting the traversals in time $O(\lg\sigma)$. Ferragina et al. [40] showed that the zero-order compression can be extended to multiary wavelet trees, reaching $nH_0(S)+o(n\lg\sigma)$ bits and time $O(1+\lg\sigma/\lg\lg n)$ for the operations, and Golynski et al. [50] reduced the space to $nH_0(S)+o(n)$ bits. Recently, Belazzougui and Navarro [12] showed that the times can be reduced to $O(1+\lg\sigma/\lg w)$, where $w=\Omega(\lg n)$ is the size of the machine word. Basically they replace the universal tables with bit-parallel operations. Their space grows to $nH_0(S)+o(n(H_0(S)+1))$. (They also prove and match the lower bound time complexity $\Theta(1+\lg(\lg\sigma/\lg w))$ using techniques that are beyond wavelet trees and this survey, but that do build on wavelet trees [7, 4].)

It should not be hard to see at this point that the sums of $n_u\lg(n_v/n_u)$ spaces used for fast upward traversals in Section 2 also add up to $(1/\epsilon)nH_0(S)$.

Changing shape. The algorithms for traversing the wavelet tree work independently of its balanced shape. Furthermore, our previous analysis of the entropy coding of the bitmap also shows that the resulting space, at least with respect to the entropy part, is independent of the shape of the tree. This was already noted by Grossi et al. [55], who proposed using the shape to optimize average query time: If we know the relative frequencies f_c with which each leaf c is sought, we can create a wavelet tree with the shape of the Huffman tree [62] of those frequencies, thus reducing the average access time to $\sum_{c\in\Sigma} f_c\lg(1/f_c)\le\lg\sigma$.

Mäkinen and Navarro [70, Sec. 3.3], instead, proposed giving the wavelet tree the Huffman shape of the frequencies with which the symbols appear in S. This has interesting consequences. First, it is easy to see that the total number of bits stored in the wavelet tree is exactly the number of bits output by a Huffman compressor that takes the symbol frequencies in S, which is upper bounded by $n(H_0(S)+1)$. Therefore, even using plain bitmap representations taking $n+o(n)$ bits of space, the total space becomes at most $n(H_0(S)+1)+o(n(H_0(S)+1))+O(\sigma\lg n)$, that is, we compress not only the data, but also the redundancy space. This may seem irrelevant compared to the $nH_0(S)+o(n)$ bits that can be obtained using Golynski et al. [50] over a balanced wavelet tree. However, it is unclear whether that approach is practical; only that of Raman et al. [93] has successful implementations [89, 28, 84], and this one leads to total space $nH_0(S)+o(n\lg\sigma)$. Furthermore, plain bitmap representations are significantly faster than compressed ones, and thus compressing the wavelet tree by giving it a Huffman shape leads to a much faster implementation in practice.

Another consequence of using Huffman shape, implied by Grossi et al. [55], is that if the accesses to the leaves are done with frequency proportional to their number of occurrences in S (which occurs, for example, if we access at random positions in S), then the average access time is $O(1+H_0(S))$, better than the $O(\lg\sigma)$ of balanced wavelet trees. A problem is that the worst case could be as bad as $O(\lg n)$ if a very infrequent symbol is sought [70]. However, one can balance wavelet subtrees after some depth, so that the average depth is

$O(1+H_0(S))$, the maximum depth is $O(\lg \sigma)$, and the total number of bits is at most $n(H_0(S)+2)$ [70].

Recently, Barbay and Navarro [10] showed that Huffman shapes can be combined with multiary wavelet trees and entropy compression of the bitmaps, to achieve space $nH_0(S)+o(n)$ bits, worst-case time $O(1+\lg \sigma / \lg \lg n)$, and average case time $O(1+H_0(S)/\lg \lg n)$.

An interesting extension of Huffman shaped wavelet trees that has not been emphasized much is to use them a mechanism to give direct access on *any* variable-length prefix-free coding. Let $S = s_1, s_2, \ldots, s_n$ be a sequence of symbols, which are encoded in some way into a bit-stream $C = c(s_1)c(s_2)\ldots c(s_n)$. For example, S may be a numeric sequence and c can be a δ-code, to favor small numbers [13], or c can be a Huffman or another prefix-free encoding. Any prefix-free encoding ensures that we can retrieve S from C, but if we want to maintain the compressed form C and access arbitrary positions of S, we need tricks like sampling S at regular intervals and store pointers to C.

Instead, a wavelet tree representation of S, where for each s_i we rather encode $c(s_i)$, uses the same number of bits of C and gives direct access to any $S[i]$ in time $O(|c(s_i)|)$. More precisely, at the bitmap root position $B_{v_{root}}[i]$ we write a 0 if $c(s_i)$ starts with a 0, and 1 otherwise. In the first case we continue by the left child and in the second case we continue by the right child, from the second bit of $c(s_i)$, until the code is exhausted. Gagie et al. [43] combined this idea with multiary wavelet trees to obtain a faster decoding.

Very recently, Grossi and Ottaviano [56] also took advantage of specific shapes, to give the wavelet tree the form of a trie of a set of strings. The goal was to handle a sequence of strings and extend operations like access and `rank` to such strings. The idea extends a previous, more limited, approach [72, 74].

High-order entropy coding. High-order compression extends zero-order compression by encoding each symbol according to a context of length k that precedes or follows it. The *k-th order empirical entropy* of S [77] is defined as $H_k(S) = \sum_{A \in \Sigma^k} (|S_A|/n)\ H_0(S_A) \leq H_{k-1}(S)$, where S_A is the string of symbols preceding context A in S. Any statistical compressor assigning fixed codes that depend on a context of length k outputs at least $nH_k(S)$ bits to encode S.

The Burrows-Wheeler transform [22] is a useful tool to achieve high-order entropy. It is a reversible transformation that permutes the symbols of a string $S[1,n]$ as follows. First sort all the suffixes $S[i,n]$ lexicographically, and then list the symbols that precede each suffix (where $S[n]$ precedes $S[1,n]$). The result, $S^{bwt}[1,n]$, is the concatenation of the strings S_A for all the contexts A. By definition, if we compress each substring S_A of S^{bwt} to its zero-order entropy, the total space is the k-th order entropy of S, for $k = |A|$.

The first [54] and second [39] reported use of wavelet trees used a similar partitioning to represent each range of S^{bwt} with a zero-order compressed wavelet tree, so as to reach $nH_k(S) + o(n \lg \sigma)$ bits of space, for any $k \leq \alpha \lg_\sigma n$ and any constant $0 < \alpha < 1$. In the second case [39], the use of S^{bwt} was explicit. The partitioning was not with a fixed context length, but instead an optimal partitioning was used [36]. This way, they obtained the given space simultaneously for any k in the range. In the first case [54], they made no reference to the Burrows-Wheeler transform, but also compressed the sequences S_A of the k-th order entropy formula, for a fixed k. We give more details on the reasons behind the use of S^{bwt} in Section 5.

Already in 2004, Grossi et al. [55] realized that the careful partitioning into many small wavelet trees, one per context, was not really necessary to achieve k-th order compression. By using a proper encoding on its bitmaps, a wavelet tree on the whole S^{bwt} could reach k-th order entropy compression of a string S. They obtained $2nH_k(S)$ bits, plus redundancy, by using γ-codes [13] on the runs of 0s and 1s in the wavelet tree bitmaps. Mäkinen and Navarro [73] observed the same fact when encoding the bitmaps using Raman et al. [93] fully indexable dictionaries. They reached $nH_k(S) + o(n \lg \sigma)$ bits of space, simultaneously for any $k \leq \alpha \lg_\sigma n$ and any constant $0 < \alpha < 1$, using just one wavelet tree for the whole string. This yielded simpler and faster indexes in practice [28].

The key property is that some entropy-compression methods are local, that is, their space is the sum of the zero-order entropies of short substrings of S^{bwt}. This can be shown to be upper-bounded by the entropy of the whole string, but also by the sum of the entropies of the substrings S_A. Even more surprisingly, Kärkkäinen and Puglisi [67] recently showed that the k-th order entropy is still reached if one cuts S^{bwt} into equally-spaced regions of appropriate length, and thus simplified these indexes further by using the faster and more practical Huffman-shaped wavelet trees on each region.

There are also more recent and systematic studies [35, 59] of the compressibility properties of wavelet trees, and how they relate to gap and run-length encodings of the bitmaps, as well to the balancing and the arity.

Exploiting repetitions. Another relevant source of compressibility is repetitiveness, that is, that $S[1, n]$ can be decomposed into a few substrings that have appeared earlier in S, or alternatively, that there is a small context-free grammar that generates S. Many compressors build on these principles [13], but supporting wavelet tree functionality on such compressed representations is harder.

Mäkinen and Navarro [71] studied the effect of repetitions in the Burrows-Wheeler transform of S. They showed that S^{bwt} could be partitioned into at most $nH_k(S)+\sigma^k$ runs of equal letters in S^{bwt}, for any k. It is not hard to see that those runs are inherited by the wavelet tree bitmaps, where run-length compression would take proper advantage of them. Mäkinen and Navarro followed a different path: they built a wavelet tree on the run heads and used a couple of bitmaps to simulate the operations on the original strings. The compressibility of those two bitmaps has been further studied by Mäkinen et al. [95, 75] in the context of highly repetitive sequence collections, and also by Simon Gog [47, Sec. 3.6.1].

In some cases, however, we need the wavelet tree of the very same string S that contains the repetition, not its Burrows-Wheeler transform. We describe such an application in the paragraph "Document retrieval indexes" of Section 6.

Recently, Navarro et al. [86] proposed a grammar-compressed wavelet tree for this problem. The key point is that repetitions in $S[1, n]$ induce repetitions in $B_{v_{root}}[1, n]$. They used Re-Pair [69], a grammar-based compressor, on the bitmaps, and enhanced a Re-Pair-based compressed sequence representation [53] to support binary `rank` (they only needed downward traversals). This time, the wavelet tree partitioning into left and right children cuts each repetition into two, so quickly after a few levels such regularities are destroyed and another type of bitmap compression (or none) is preferred. While the theoretical space analysis is too weak to be useful, the result is good in practice and leaves open the challenge of achieving stronger theoretical and practical results.

We will find even more specific wavelet tree compression problems later.

4 Sequences, Reorderings, or Point Grids?

Now that we have established the basic structure, operations, and encodings of wavelet trees, let us take a view with more perspective. Various applications we have mentioned display different ways to regard a wavelet tree representation.

As a sequence of values. This is the most basic one. The wavelet tree on a sequence $S = s_1, \ldots, s_n$ represents the values s_i. The most important operations that the wavelet tree must offer to support this view are, apart from accessing any $S[i]$ (that we already explained in Section 2), `rank` and `select` on S. For example, the second main usage of wavelet trees [39, 40] used access and `rank` on the wavelet tree built on sequence S^{bwt} in order to support searches on S.

The process to support $\texttt{rank}_c(S,i)$ is similar to that for access, with a subtle difference. We start at position i in $B_{v_{root}}$, and decide whether to go left or right depending on where is the leaf corresponding to c (and not depending on $B_{v_{root}}[i]$). If we go left, we rewrite $i \leftarrow \texttt{rank}_0(B_{v_{root}}, i)$, else we rewrite $i \leftarrow \texttt{rank}_1(B_{v_{root}}, i)$. When we arrive at the leaf c, the value of i is the final answer. The time complexity for this operation is that of a downward traversal towards the leaf labeled c. To support $\texttt{select}_c(S,i)$ we just apply the upward tracking, as described in Section 2, starting at the i-th position of the leaf labeled c.

As a reordering. Less obviously, the wavelet tree structure describes a stable ordering of the symbols in S, so that if one traverses the leaves one finds first all the occurrences of the smaller symbols, and within the same symbol (i.e., the same leaf), they are ordered by original position. As it will be clear in Section 5, one can argue that this is the usage of wavelet trees made by their creators [54].

In this case, tracking a position downwards in the wavelet tree tells where it goes after sorting, and tracking a position upwards tells where each symbol is placed in the sequence. An obvious application is to encode a permutation π over $[1..n]$. Our best wavelet tree takes $n \lg n + o(n)$ bits and can compute any $\pi(i)$ and $\pi^{-1}(i)$ in time $O(\lg n / \lg \lg n)$ by carrying out, respectively, downward and upward tracking of position i. We will see improvements on this idea later.

As a grid of points. The slightly less general structure of Chazelle [25] can be taken as the representation of a set of points supported by wavelet trees. It is generally assumed that we have an $n \times n$ grid with n points so that no two points share the same row or column (i.e., a permutation). A general set of n points is mapped to such a discrete grid by storing the real coordinates somewhere else and breaking ties somehow (arbitrarily is fine in most cases).

Take the set of points (x_i, y_i), in x-coordinate order (i.e., $x_i < x_{i+1}$). Now define string $S[1,n] = y_1, y_2, \ldots, y_n$. Then we can find the i-th point in x-coordinate order by accessing $S[i]$. Moreover, since the wavelet tree is representing the reordering of the points according to y-coordinate, one can find the i-th point in y-coordinate order by tracking upwards the i-th point in the leaves.

Unlike permutations, here the emphasis is in counting and reporting the points that lie within a rectangle $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$. This is solved through a more complicated tracking mechanism, well-known in computational geometry and also described explicitly on wavelet trees [72]. We start at the root bitmap range $B_{v_{root}}[x_l, x_r]$, where $x_l = x_{min}$ and $x_r = x_{max}$. Now we map the interval to the left *and* to the right, using $x_l \leftarrow \texttt{rank}_{0/1}(B_{v_{root}}, x_l - 1) + 1$ and $x_r \leftarrow \texttt{rank}_{0/1}(B_{v_{root}}, x_r)$, and continue recursively. At any node along the recursion, we may stop if (*i*) the interval $[x_l, x_r]$ becomes empty (thus there are no points to report); (*ii*) the interval of leaves (i.e., y-coordinate values) represented by the node has no intersection with $[y_{min}, y_{max}]$; (*iii*) the interval of leaves is contained in $[y_{min}, y_{max}]$. In case (*iii*) we can count the number of points falling in this sub-rectangle as $x_r - x_l + 1$. As it is well known that we visit only $O(\lg n)$ wavelet tree nodes before stopping all the recursive calls (see, e.g., a recent detailed proof, among other more sophisticated wavelet tree properties [45]), the counting time is $O(\lg n)$. Each of the $x_r - x_l + 1$ points found in each node can be tracked up and down to find their x- and y-coordinates, in $O(\lg n)$ time per reported occurrence. There are more efficient variants of this technique that we will cover in Section 7, but they build on this basic idea.

5 Applications as Sequences

Full-text indexes. A full-text index built a string $S[1,n]$ is able to count and locate the occurrences of arbitrary patterns $P[1,m]$ in S. A classical index is the *suffix array* [52, 76], $A[1,n]$, which lists the starting positions of all the suffixes of S, $S[A[i], n]$, in lexicographic order, using $n\lceil \lg n \rceil$ bits. The starting positions of the occurrences of P in S appear in a contiguous range in A, which can be binary searched in time $O(m \lg n)$, or $O(m + \lg n)$ by doubling the space. A *suffix tree* [98, 78, 1] is a more space-consuming structure (yet still $O(n \lg n)$ bits) that can find the range in time $O(m)$. After finding the range, each occurrence is reported in constant time, both in suffix trees and arrays.

The suffix array of S is closely related to its Burrows-Wheeler transform: $S^{bwt}[i] = S[A[i]-1]$ (taking $S[0] = S[n]$). Ferragina and Manzini [37, 38] showed how, using at most $2m$ access and $\texttt{rank}$ operations on S^{bwt}, one could count the number of occurrences in S of a pattern $P[1,m]$. Using multiary wavelet trees [40, 50] this gives a counting time of $O(m)$ on polylog-sized alphabets, and $O(m \lg \sigma / \lg \lg n)$ in general. Each such occurrence can then be located in time $O(\lg^{1+\epsilon} n \lg \sigma / \lg \lg n)$ for any $\epsilon > 0$, at the price of $O(n / \lg^{\epsilon} n) = o(n)$ further bits of space. This result has been superseded very recently [7, 12, 11, 4], in some cases using wavelet trees as a part of the solution, and in all cases with some extra price in terms of redundancy, such as $o(nH_k(S))$ and $O(n)$ further bits.

Grossi et al. [57, 58, 54] used wavelet trees to obtain a similar result via a quite different strategy. They represented A by means of a permutation $\Psi(i) = A^{-1}[A[i]+1]$, that is, the cell in A pointing to $A[i]+1$. Ψ turns out to be formed by σ contiguous ascending runs. The suffix array search can be simulated in $O(m \lg n)$ accesses to Ψ. They encode Ψ separately for the range of each context S_A (recall paragraph "High-order entropy coding" in Section 3). As all the Ψ pointers coming from each run are increasing, a wavelet tree is used to describe how the σ ascending sequences of pointers coming from each run are intermingled in the range of S_A. This turns out to be, precisely, the wavelet tree of S_A. This is why both Ferragina et al. and Grossi et al. obtain basically the same space, $nH_k(S) + o(n \lg \sigma)$ bits. Due to the different search strategy, the counting time of Grossi et al. is higher. On the other hand, the representation of Ψ allows them to locate patterns in sublogarithmic time, still using $O(nH_k(S)) + o(n \lg \sigma)$ bits.

This is the best known usage of wavelet trees as sequences, and it is well covered in earlier surveys [82]. New extensions of these basic concepts, supporting more sophisticated search problems, appear every year (e.g., [94, 14]). We cover next other completely different applications.

Positional inverted indexes. Consider a natural language text collection. A positional inverted index is a data structure that stores, for each word, the list of the positions where it appears in the collection [3]. In compressed form [99] it takes space close to the zero-order entropy of the text *seen as a sequence of words* [82]. This entropy yields very competitive compression in natural language texts. Yet, we need to store both the text (usually zero-order compressed, so that direct access is possible) and the inverted index, adding up to at least $2nH_0(S)$, where S is the text regarded as a sequence of word identifiers. Inverted indexes are by far the most popular data structures to index natural language text collections, so reducing their space requirements is of high relevance.

By representing the sequence of word identifiers using a wavelet tree, we obtain a single representation for both the text and the inverted index, all within $nH_0(S) + o(n)$ bits [28]. In order to access any text word, we just compute $S[i]$. In order to access the i-th element of the inverted list of any word c, we compute $\texttt{select}_c(S, i)$. Furthermore, operation $\texttt{rank}_c(S, i)$ is useful to implement some list intersection algorithms [8], as it finds the position i in the inverted list of word c more efficiently than with a binary or exponential search.

Arroyuelo et al. [2] extended this functionality to *document retrieval*: retrieve the distinct documents where a word appears. They use a special symbol "\$" to mark document boundaries. Then, given the first occurrence of a word c, $p = \texttt{select}_c(S, 1)$, the document where this occurrence lies is $j = \texttt{rank}_\$(S, p)+1$, document j ends at position $p' = \texttt{select}_\$(S, j)$, it contains $o = \texttt{rank}_c(S, p, p')$ occurrences of the word c, and the search for further relevant documents can continue from query $\texttt{select}_c(S, o + 1)$.

An improvement over the basic idea is to use multiary wavelet trees, more precisely of arity up to 256, and using the property that wavelet trees give direct access to any variable-length code. Brisaboa et al. [19] started with a byte-oriented encoding of the text words (using either Huffman with 256 target symbols, or other practical encoding methods [20]) and then organized the sequence of codes into a wavelet tree, as described in the paragraph "Changing shape" of Section 3. A naive byte-based `rank` and `select` implementation on the wavelet tree levels gives good results in this application, with the bytes represented in plain form. The resulting structure is indeed competitive with positional inverted indexes in many cases. A variant specialized on XML text collections, where the codes are also used to distinguish structural elements (tags, content, attributes, etc.) in order to support some XPath queries, is also being developed [18].

Graphs. Another simple application of this idea is the representation of directed graphs [28]. Let G be a graph with n nodes and e edges. An adjacency list, using $n \lg e + e \lg n$ bits (the n pointers to the lists plus the e target nodes) gives direct access to the neighbors of any node v. If we want also to perform reverse nagivation, that is, to know which nodes point to v, we must spend other $n \lg e + e \lg n$ bits to represent the transposed graph.

Once again, representing with a wavelet tree the sequence $S[1, e]$ concatenating all the adjacency lists, plus a compressed bitmap $B[1, e]$ marking the beginnings of the lists, gives access to both types of neighbors within space $n \lg(e/n) + e \lg n + O(n) + o(e)$, which is close to the space of the plain representation (actually, possibly less). To retrieve the i-th neighbor of a node v, we compute the starting point of the list of v, $l \leftarrow \mathtt{select}_1(B, v)$, and then access $S[l + i - 1]$. To retrieve the i-th reverse neighbor of a node v, we compute $p \leftarrow \mathtt{select}_v(S, i)$ to find the i-th time that v is mentioned in an adjacency list, and then compute with $\mathtt{rank}_1(B, p)$ the owner of the list where v is mentioned. Both operations take time $O(\lg n / \lg \lg n)$. This is also useful to represent undirected graphs, where adjacency lists must usually represent each edge twice. With a wavelet tree we can choose any direction for an edge, and at query time we join direct and reverse neighbors of nodes to build their list.

Note, finally, that the wavelet tree can compress S to its zero-order entropy, which corresponds to the distribution of in-degrees of the nodes. A more sophisticated variant of this idea, combined with Re-Pair compression [69], was shown to be competitive with current Web graph compression methods [29].

6 Applications as Reorderings

Apart from its first usage [54], that can be regarded as encoding a reordering, wavelet trees offer various interesting applications when seen in this way.

Permutations. As explained in Section 4, one can easily encode a permutation with a wavelet tree. It is more interesting that the encoding can take less space when the permutation is, in a sense, compressible. Barbay and Navarro [9, 10] considered permutations π of $[1..n]$ that can be decomposed into ρ contiguous ascending runs, of lengths $r_1, r_2, \ldots, r_\rho$. They define the entropy of such a permutation as $H(\pi) = \sum_{i=1}^{\rho} (r_i/n) \lg(n/r_i)$, and show that it is possible to sort an array with such ascending runs in time $O(n(H(\pi) + 1))$. This is obtained by building a Huffman tree on the run lengths (seen as frequencies) and running a mergesort-like algorithm that follows the Huffman tree shape.

They note that, if we encode with 0 or 1 the results of the comparisons of the mergesort algorithm at each node of the merging tree, the resulting structure contains at most $n(H(\pi)+1)$ bits, and it represents the permutation. Starting at position i in the top bitmap $B_{v_{root}}$ one can track down the position exactly as done with wavelet trees, so as to arrive at position j of the t-th leaf (i.e., run). By storing, in $O(\rho \lg n)$ bits, the starting position of each run in π, we can convert the leaf position into a position in π. Therefore the downward traversal solves operation $\pi^{-1}(i)$, because it starts from value i (i.e., position i after sorting π), and gives the position in π from where it started before the merging took place. The corresponding upward traversal, consequently, solves $\pi(i)$. Other types of runs, more and less general, are also studied [9, 10].

Some thought reveals that this structure is indeed the wavelet tree of a sequence formed by replacing, in π^{-1}, each symbol belonging to the i-th run, by the run identifier i. Then the fact that a downward traversal yields $\pi^{-1}(i)$ and that the upward traversal yields $\pi(i)$ are natural consequences. This relation is made more explicit in a later article [7, 4].

Generic numeric sequences. There are several basic problems on sequences of numbers that can be solved in nontrivial ways using wavelet trees. We mention a few that have received attention in the literature.

One such problem is the *range quantile query*: Preprocess a sequence of numbers $S[1,n]$ on the domain $[1..\sigma]$ so that later, given a range $[l,r]$ and a value i, we can compute the i-th smallest element in $S[l,r]$.

Classical solutions to this problem have used nearly quadratic space and constant time. Only a very recent solution [65] reaches $O(n \lg n)$ bits of space (apart from storing S) and $O(\lg n / \lg\lg n)$ time. We show that, by representing S with a wavelet tree, we can solve the problem in $O(\lg \sigma)$ time and just $o(n)$ extra bits [46, 45]. This is close to $O(\lg n / \lg\lg n)$ (in this problem, we can always make $\sigma \le n$ hold), and it can be even better if σ is small compared to n.

Starting from the range $S[l,r]$, we compute $\texttt{rank}_0(B_{v_{root}}, l, r)$. If this is i or more, then the i-th value in this range is stored in the left subtree, so we go to the left child and remap the interval $[l,r]$ as done for counting points in a range (see Section 4). Otherwise we go right, subtracting $\texttt{rank}_0(B_{v_{root}}, l, r)$ from i and remapping $[l,r]$ in the same way. When we arrive at a leaf, its label is the i-th smallest element in $S[l,r]$.

Another fundamental problem is called *range next value*: Preprocess a sequence of numbers $S[1,n]$ on the domain $[1..\sigma]$ so that later, given a range $[l,r]$ and a value x, we return the smallest value in $S[l,r]$ that is larger than x.

The state of the art also includes superlinear-space and constant-time solutions, as well as one using $O(n \lg n)$ bits of space and $O(\lg n / \lg\lg n)$ time [100]. Once again, we achieve $o(n)$ extra bits and $O(\lg \sigma)$ time using wavelet trees [45] (we improve this time in the paragraph "Binary relations" of Section 7).

Starting at the root from the range $S[l,r]$, we see if value x labels a leaf descending from the left or from the right child. If x descends from the right child, then no value on the left child can be useful, so we recursively descend to the right child and remap the interval $[l,r]$ as done for counting points in a range. Else, there may be values $> x$ on both children, but we prefer those on the left, if any. So we first descend to the left child looking for an answer (there may be no answer if, at some node, the interval $[l, r]$ becomes empty). If the left child returns an answer, this is what we seek and we return it. If, however, there is no value $> x$ on the left child, we seek the smallest value on the right child. We then enter into another mode where we see if there is any 0-bit in $B_v[l, r]$. If there is one, we go to the left child, else we go to the right child. It can be shown that the overall process takes $O(\lg \sigma)$ time.

A variant of the range next value problem is called *prevLess* [68]: return the rightmost value in $S[1, r]$ that is smaller than x. Here we start with $S[1, r]$. If value x labels a leaf descending from the left, we map the interval to the left child and continue recursively from there. If, instead, x descends from the right child, then the answer may be on the left or the right child, and we prefer the rightmost in $[1, r]$. Any 0-bit in $B_v[1, r]$ is a value smaller than x and thus a valid answer. We use `rank` and `select` to find the rightmost 0 in $B_v[1, r]$. We also continue recursively by the right child, and if it returns an answer, we map it to the bitmap $B_v[1, r]$. Then we choose the rightmost between the answer from the right child and the rightmost zero. The overall time is $O(\lg \sigma)$.

Non-positional inverted indexes. These indexes store only the list of distinct documents where each word appears, and come in two flavors [99, 3]. In the first, the documents for each word are sorted by increasing identifier. This is useful to implement list unions and intersections for boolean, phrase and proximity queries. In the second, a "weight" (measuring importance somehow) is assigned to each document where a word appears. The lists of each word store those weights and are sorted by decreasing weight. This is useful to implement ranked bag-of-word queries, which give the documents with highest weights added over all the query words. It would seem that, unless one stores two inverted indexes, one must choose one order in detriment of the queries of the other type.

By representing a reordering, wavelet trees can store both orderings simultaneously [85, 45]. Let us represent the documents where each word appears in decreasing weight order, and concatenate all the lists into a sequence $S[1, n]$. A bitmap $B[1, n]$ marks the starting positions of the lists, and the weights are stored separately. Then, a wavelet tree representation of S simulates, within the space of just one list, both orderings. By accessing $S[l+i-1]$, where $l = \mathtt{select}_1(B, c)$, we obtain the i-th element of the inverted list of word c, in decreasing weight order. To access the i-th element of the inverted list of a word in increasing document order, we also compute the end of its list, $r = \mathtt{select}_1(B, c+1) - 1$, and then run a range quantile query for the i-th smallest value in the range $[l, r]$. Many other operations of interest in information retrieval can be carried out with this representation and little auxiliary data [85, 45].

Document retrieval indexes. An interesting extension to full-text retrieval is document retrieval, where a collection $S[1, n]$ of general strings (so inverted indexes cannot be used) is to be indexed to answer different document retrieval queries. The most basic one, document listing, is to output the distinct documents where a pattern $P[1, m]$ appears. Muthukrishnan [80] defined a so-called *document array* $D[1,n]$, where $D[i]$ gives the document to which the i-th lexicographically smallest suffix of S belongs (i.e., where the suffix $S[A[i],n]$ belongs, where A is the suffix array of S). He also defined an array $C[1,n]$, where $C[i]$ points to the previous occurrence of $D[i]$ in D. A suffix tree was used to identify the range $A[l,r]$ of the pattern occurrences, so that we seek to report the distinct elements in $D[l,r]$. With further structures to find minima in ranges of C [15], Muthukrishnan gave an $O(m+\mathit{occ})$ algorithm to find the occ distinct documents where P appears. This is time-optimal, yet the space is impractical.

This is another case where wavelet trees proved extremely useful. Mäkinen and Välimäki [97] showed that, if one implemented D as a wavelet tree, then array C was not necessary, since $C[i] = \mathtt{select}_{D[i]}(D, \mathtt{rank}_{D[i]}(D, i-1))$. They also used a compressed full-text index [39] to identify the range $D[l,r]$, so the total time turned out to be $O(m \lg \sigma + \mathit{occ} \lg d)$, where d is the number of documents in S. Moreover, for each document c output, $\mathtt{rank}_c(D,l,r)$ gave the number of times P appeared in c, which is important for ranked document retrieval.

Gagie et al. [46, 45] showed that an application of range quantile queries enabled the wavelet tree to solve this problem elegantly and without any range minima structure: The first distinct document is the smallest value in $D[l,r]$. If it occurs f_1 times, then the second distinct document is the $(1+f_1)$-th smallest value in $D[l,r]$, and so on. They retained the complexities of Mäkinen and Välimäki, but the solution used less space and time in practice. Later [45] they replaced the range quantile queries by a depth-first traversal of the wavelet tree that reduced the time complexity, after the suffix array search, to $O(\mathit{occ} \lg(d/\mathit{occ}))$. The technique is similar to the two-dimensional range searches: recursively enter into every wavelet tree branch where the mapped interval $[l,r]$ is not empty, and report the leaves found, with frequency $r-l+1$.

This depth-first search method can easily be extended to support more complex queries, for example t-thresholded ones: given s patterns, we want the documents where at least t of the terms appear. We can first identify the s ranges in D and then traverse the wavelet tree while maintaining the s ranges, stopping when less than t intervals are nonempty, or when we arrive at leaves (where we report the document). Other sophisticated traversals have been proposed for retrieving the documents ranked by number of occurrences of the patterns [33].

An interesting problem is how to compress the wavelet tree of D effectively. The zero-order entropy of D has to do with document lengths, which is generally uninteresting, and unrelated to the compressiblity of S. It has been shown [44, 86] that the compressibility of S shows up as repetitions in D, which has stimulated the development of wavelet tree compression methods that take advantage of the repetitiveness of D, as described at the end of Section 3.

7 Applications as Grids

Discrete grids. Much work has been done in Computational Geometry over structures very similar to wavelet trees. We only highlight some results of interest, generally focusing on structures that use linear space. We assume here that we have an $n \times n$ grid with n points not sharing rows nor columns. Interestingly, these grids with range counting and reporting operations have been intensively used in compressed text indexing data structures [66, 81, 38, 72, 26, 16, 30, 68]

Range counting can be done in time $O(\lg n/\lg\lg n)$ and $O(n \lg n)$ bits [64]. This time cannot be improved within space $O(n \lg^{O(1)} n)$ [90], but it can be matched with a multiary wavelet-tree like structure using just $n \lg n + o(n \lg n)$ bits [16]. Reaching this time, instead of the easy $O(\lg n)$ we have explained in Section 4, requires a sophisticated solution to the problem of doing the range counting among several consecutive children of a node, that are completely contained in the x-range of the query. They [16] also obtain a range reporting time (for the *occ* points in the range) of $O((1+occ)\lg n/\lg\lg n)$. This is not surprising once counting has been solved: it is a matter of upward or downward tracking on a multiary wavelet tree. The technique for faster upward tracking we described in the paragraph "Speeding up traversals" of Section 2 can be used to improve the reporting time to $O((1+occ)\lg^{\epsilon} n)$, using $O((1/\epsilon)n \lg n)$ bits of space [24].

Wavelet trees offer relevant solutions to other geometric problems, such as finding the dominant points in a grid, or solving visiblity queries. Those problems can be recast as a sequence of queries of the form "find the smallest element larger than x in a range", described in the paragraph "Generic numeric sequences" of Section 6, and therefore solved in time $O(\lg n)$ per point retrieved [83]. That paper [83, 87] also studies extensions of geometric queries where the points have weights and statistical queries on them are posed, such as finding range sums, averages, minima, quantiles, majorities, and so on. The way those queries are solved open interesting new avenues in the use of wavelet trees.

Some queries, such as finding the minimum value of a two-dimensional range, are solved by enriching wavelet trees with extra information aligned to the bitmaps. Recall that each wavelet tree node v handles a subsequence S_v of the sequence of points $S[1, n]$. To each node v with bitmap $B_v[1, n_v]$ we associate a data structure using $2n_v + o(n_v)$ bits that answers one-dimensional range minimum queries [41] on $S_v[1, n_v]$. Once built, this structure does not need to access S_v, yet it gives the position of the minimum in constant time. Since, as explained, a two-dimensional range is covered by $O(\lg n)$ wavelet tree nodes, only those $O(\lg n)$ minima must be tracked upwards, where the actual weights are stored, to obtain the final result. Thus the query requires $O(\lg^{1+\epsilon} n)$ time and $O((1/\epsilon)n \lg n)$ bits of space by using the fast upward tracking mechanism.

Other queries, such as finding the i-th smallest value of a two-dimensional range, are handled with a wavelet tree *on the weight values*. Each wavelet tree node stores a grid with the points whose weights are in the range handled by that node. Then, by doing range counting queries on those grids, one can descend left or right, looking for the rightmost leaf (i.e., value) such that the counts of the children to the left of the path followed add up to less than i. The total time is $O(\lg^2 n/\lg\lg n)$, however the space becomes superlinear, $O(n \lg^2 n)$ bits.

Finally, an interesting extension to the typical point grids are grids of rectangles, which are used in geographic information systems as minimum bounding rectangles of complex objects. Then one wishes to find the set of rectangles that intersect a query rectangle. This is well solved with an R-tree data structure [60], but a wavelet tree may offer interesting space reductions. Brisaboa et al. [21] describe a technique to store n rectangles where one does not contain another in the x-coordinate range (so the set is first separated into maximal "x-independent" subsets and each subset is queried separately). Two arrays with the ascending lower and upper x-coordinates of the rectangles are stored (as the sets are x-independent, the same position in both arrays corresponds to the same rectangle). A wavelet tree on those x-coordinate-sorted rectangles is set up, so that each node handles a range of y-coordinate values. This wavelet tree stores two bitmaps per node v: one tells whether the rectangle $S_v[i]$ extends to the y-range of the left child, and the other whether it extends to the right child. Both bitmaps can store a 1 at a position i, and thus the rectangle is stored in both subtrees. To avoid representing a large rectangle more than $O(\lg n)$ times, both bits are set to 0 (which is otherwise impossible) when the rectangle completely contains the y-range of the current node. The total space is $O(n \lg n)$ bits.

Given a query $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$, we look for x_{min} in the array of *upper* x-coordinates, to find position x_l, and look for x_{max} in the array of *lower* x-coordinates, to find position x_r. This is because a query intersects a rectangle on the x-axis if the query does not start after the rectangle ends and the query does not end before the rectangle starts. Now the range $[x_l, x_r]$ is used to traverse the wavelet tree almost like on a typical range search, except that we map to the left child using $\texttt{rank}_1$ on one bitmap, and to the right child using $\texttt{rank}_1$ on the other bitmap. Furthermore, we report all the rectangles where both bitmaps contain a 0-bit, and we remove duplicates by merging results at each node, as the same rectangle can be encountered several times. The overall time to report the *occ* rectangles is still $O((1 + occ) \lg n)$.

Binary relations. A binary relation R between two sets A and B can be thought of as a grid of size $|A| \times |B|$, containing $|R|$ points. Apart from strings, permutations and our grids, that are particular cases, binary relations are good abstractions for a large number of more applied structures. For example, a non-positional inverted index is a binary relation between a set of words and a set of documents, so that a word is related to the documents where it appears. As another example, a graph is a binary relation between the set of nodes and itself.

The most typical operations on binary relations are determining the elements $b \in B$ that are related to some $a \in A$ and vice versa, and determining whether a pair $(a, b) \in A \times B$ is related in R. However, more complex queries are also of interest. For example, counting or retrieving the documents related to any term in a range enables on-the-fly stemming and query expansion. Retrieving the terms associated to a document permits vocabulary analyses. Accessing the documents in a range related to a term enables searches local to subcollections. Range counting and reporting allows regarding graphs at a larger granularity (e.g., a Web graph can be regarded as a graph of hosts, or of pages, on the fly).

Barbay et al. [5, 6] studied a large number of complex queries for binary relations, including accessing the points in a range in various orders, as well as reporting rows or columns containing points in a range. They proposed two wavelet-tree-like data structures for handling the operations. One is basically a wavelet tree of the set of points (plus a bitmap that indicates when we move from one column to the next). It turns out that almost all the solutions described so far on wavelet trees find application to solve some of the operations.

In the extended version [6] they use multiary wavelet trees to reduce the times of most of the operations. Several nontrivial structures and algorithms are designed in order to divide the times of various operations by $\lg \lg n$ (the only precedent we know of is that of counting the number of points in a range [16]). For example, it is shown how to solve the *range next value* problem (recall paragraph "Generic numeric sequences" of Section 6) in time $O(\lg n / \lg \lg n)$. Others, like the *range quantile query*, stay no better than $O(\lg n)$.

Barbay et al. also propose a second data structure that is analogous to the one described for rectangles in the paragraph "Discrete grids". Two bitmaps are stored per node, indicating whether a given column has points in the first and in the second range of rows. This extension of a wavelet tree is less powerful than the previous structure, but it is shown that its space is close to the *entropy* of the binary relation: $(1+\sqrt{2})H+O(|A|+|B|+|R|)$ bits, where $H = \lg \binom{|A|\cdot|B|}{|R|}$. This is not achieved with the classical wavelet tree. A separate work [34] builds on this to obtain a fully-compressed grid representation, within $H + o(H)$ bits.

Colored range queries. A problem of interest in areas like query log and web mining is to count the different *colors* in a sequence $S[1, n]$ over a universe of σ colors. Inspired in the idea of Muthukrishnan [80] for document retrieval (recall paragraph "Document retrieval indexes" in Section 6), Gagie et al. [44] showed that this is a matter of counting how many values smaller than l are there in $C[l, r]$, where $C[i] = \max\{j<i, S[j]=S[i]\}$. This is a range counting query for $[l, r] \times [1, l-1]$ on C seen as a grid, that can be solved in time $O(\lg n)$ using the wavelet tree of C. Note that this wavelet tree, unlike that of S, uses $n \lg n + o(n)$ bits. Gagie et al. compressed it to $n \lg \sigma + O(n \lg \lg n)$ bits, by taking advantage of the particular structure of C, which shows up in the bit-vectors. Gagie and Kärkkäinen [42] then reduced the space to $nH_0(S) + o(nH_0(S)) + O(n)$ with more advanced techniques, and also reduced the query time to $O(\lg(r - l + 1))$.

8 Conclusions and Further Challenges

We have described the wavelet tree, a surprisingly versatile data structure that offers nontrivial solutions to a wide range of problems in areas like string processing, computational geometry, and many more. An important additional asset of the wavelet tree is its simplicity to understand, teach, and program. This makes it a good data structure to be introduced at an undegraduate level, at least in its more basic variants. In many cases, solutions with better time complexity than the ones offered by wavelet trees are not so practical nor easy to implement.

Wavelet trees seem to be unable to reach access and `rank`/`select` times of the form $O(\lg \lg \sigma)$, as other structures for representing sequences do [51], close to the lower bounds [12]. However, both have been combined to offer those time complexities and good zero-order compression of data and redundancy [7, 4, 12].

Yet, the lower bounds on some geometric problems [24], matched with current wavelet trees [16, 6], suggest that this combination cannot be carried out much further than those three operations. Still, there are some complex operations where it is not clear that wavelet trees have matched lower bounds [45].

We have described the wavelet tree as a static data structure. However, if the bitmaps or sequences stored at the nodes support insertions and deletions in time $indel(n)$, then the wavelet tree easily supports insertions and deletions in the sequence $S[1, n]$ it represents, in time $O(h \cdot indel(n))$, where h is its height. This has been used to support indels in time $O((1 + \lg \sigma / \lg \lg n) \lg n / \lg \lg n)$ [61, 88]. The alphabet, however, is still fixed in those solutions. While such a limitation may seem natural for sequences, it looks definitely artificial when representing grids: one can insert and delete new x-coordinates and points, but the y-coordinate universe cannot change. Creating or removing alphabet symbols requires changing the shape of the wavelet tree, and the bitmaps or sequences stored at the nodes undergo extensive modifications upon small tree shape changes (e.g., AVL rotations). Extending dynamism to support this type of updates, with good time complexities at least in the amortized sense, is an important challenge for this data structure. It is also unclear what is the dynamic lower bound on a general alphabet; on a constant-size alphabet it is $\Theta(\lg n / \lg \lg n)$ [23]. Very recently [56] a dynamic scheme for a particular case (sequences of strings) has been proposed.

A path that, in our opinion, has only started to be exploited, is to enhance the wavelet tree with "one-dimensional" data structures at its nodes v, so that, by efficiently solving some kind of query over the corresponding subsequences S_v, we solve a more complex query on the original sequence S. In most cases along this survey, these one-dimensional queries have been `rank` and `select` on the bitmaps, but we have already shown some examples involving more complicated queries [44, 87, 83]. This approach may prove to be very fruitful.

In terms of practice, although there are many successful and publicly available implementations of wavelet tree variants (see, e.g., `libcds.recoded.cl` and `http://www.uni-ulm.de/in/theo/research/sdsl.html`), there are some challenges ahead, such as carrying to practice the theoretical results that promise fast and small multiary wavelet trees [40, 50, 17] and lower redundancies [49, 91, 50].

Acknowledgements. Thanks to Jérémy Barbay, Ana Cerdeira, Travis Gagie, Juha Kärkkäinen, Susana Ladra, Simon Puglisi, and all those who took the time to read early versions of this manuscript.

References

1. A. Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO ISI Series, pages 85–96. Springer-Verlag, 1985.
2. D. Arroyuelo, S. González, and M. Oyarzún. Compressed self-indices supporting conjunctive queries on document collections. In *Proc. 17th SPIRE*, pages 43–54, 2010.
3. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley, 2nd edition, 2011.

4. J. Barbay, F. Claude, T. Gagie, G. Navarro, and Y. Nekrich. Efficient fully-compressed sequence representations. *CoRR*, abs/0911.4981v4, 2012.
5. J. Barbay, F. Claude, and G. Navarro. Compact rich-functional binary relation representations. In *Proc. 9th LATIN*, pages 170–183, 2010.
6. J. Barbay, F. Claude, and G. Navarro. Compact binary relation representations with rich functionality. *CoRR*, abs/1201.3602, 2012.
7. J. Barbay, T. Gagie, G. Navarro, and Y. Nekrich. Alphabet partitioning for compressed rank/select and applications. In *Proc. 21st ISAAC*, pages 315–326 (part II), 2010.
8. J. Barbay, A. López-Ortiz, T. Lu, and A. Salinger. An experimental investigation of set intersection algorithms for text searching. *ACM J. Exp. Alg.*, 14, 2009.
9. J. Barbay and G. Navarro. Compressed representations of permutations, and applications. In *Proc. 26th STACS*, pages 111–122, 2009.
10. J. Barbay and G. Navarro. On compressing permutations and adaptive sorting. *CoRR*, abs/1108.4408, 2011.
11. D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. In *Proc. 19th ESA*, pages 748–759, 2011.
12. D. Belazzougui and G. Navarro. New lower and upper bounds for representing sequences. *CoRR*, abs/1111.2621, 2011.
13. T. Bell, J. Cleary, and I. Witten. *Text Compression.* Prentice Hall, 1990.
14. T. Beller, S. Gog, E. Ohlebusch, and T. Schnattinger. Computing the longest common prefix array based on the Burrows-Wheeler transform. In *Proc. 18th SPIRE*, pages 197–208, 2011.
15. M. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. 4th LATIN*, pages 88–94, 2000.
16. P. Bose, M. He, A. Maheshwari, and P. Morin. Succinct orthogonal range search structures on a grid with applications to text indexing. In *Proc. 11th WADS*, pages 98–109, 2009.
17. A. Bowe. *Multiary Wavelet Trees in Practice.* Honours thesis, RMIT Univ., Australia, 2010.
18. N. Brisaboa, A. Cerdeira, and G. Navarro. A compressed self-indexed representation of XML documents. In *Proc. 13th ECDL*, pages 273–284, 2009.
19. N. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Reorganizing compressed text. In *Proc. 31st SIGIR*, pages 139–146, 2008.
20. N. Brisaboa, A. Fariña, G. Navarro, and J. Paramá. Lightweight natural language text compression. *Inf. Retr.*, 10:1–33, 2007.
21. N. Brisaboa, M. Luaces, G. Navarro, and D. Seco. A fun application of compact data structures to indexing geographic data. In *Proc. 5th FUN*, pages 77–88, 2010.
22. M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation, 1994.
23. H.-L. Chan, W.-K. Hon, T.-W. Lam, and K. Sadakane. Compressed indexes for dynamic text collections. *ACM Trans. Alg.*, 3(2):article 21, 2007.
24. T. Chan, K. G. Larsen, and M. Pătraşcu. Orthogonal range searching on the RAM, revisited. In *Proc. 27th SoCG*, pages 1–10, 2011.
25. B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comp.*, 17(3):427–462, 1988.
26. Y.-F. Chien, W.-K. Hon, R. Shah, and J. Vitter. Geometric Burrows-Wheeler transform: Linking range searching and text indexing. In *Proc. 18th DCC*, pages 252–261, 2008.

27. D. Clark. *Compact Pat Trees.* PhD thesis, Univ. of Waterloo, Canada, 1996.
28. F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. 15th SPIRE*, pages 176–187, 2008.
29. F. Claude and G. Navarro. Extended compact Web graph representations. In *Algorithms and Applications (Ukkonen Festschrift)*, pages 77–91, 2010.
30. F. Claude and G. Navarro. Self-indexed grammar-based compression. *Fund. Inf.*, 111(3):313–337, 2010.
31. F. Claude, P. Nicholson, and D. Seco. Space efficient wavelet tree construction. In *Proc. 18th SPIRE*, pages 185–196, 2011.
32. T. Cover and J. Thomas. *Elements of Information Theory.* Wiley, 1991.
33. J. S. Culpepper, G. Navarro, S. J. Puglisi, and A. Turpin. Top-k ranked document search in general text databases. In *Proc. 18th ESA*, pages 194–205 (part II), 2010.
34. A. Farzan, T. Gagie, and G. Navarro. Entropy-bounded representation of point grids. In *Proc. 21st ISAAC*, pages 327–338 (part II), 2010.
35. P. Ferragina, R. Giancarlo, and G. Manzini. The myriad virtues of wavelet trees. *Inf. Comp.*, 207(8):849–866, 2009.
36. P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. *J. ACM*, 52(4):688–713, 2005.
37. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st FOCS*, pages 390–398, 2000.
38. P. Ferragina and G. Manzini. Indexing compressed texts. *J. ACM*, 52(4):552–581, 2005.
39. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly FM-index. In *Proc. 11th SPIRE*, pages 150–160, 2004.
40. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Alg.*, 3(2):article 20, 2007.
41. J. Fischer. Optimal succinctness for range minimum queries. In *Proc. 9th LATIN*, pages 158–169, 2010.
42. T. Gagie and J. Kärkkäinen. Counting colours in compressed strings. In *Proc. 22nd CPM*, pages 197–207, 2011.
43. T. Gagie, G. Navarro, and Y. Nekrich. Fast and compact prefix codes. In *Proc. 36th SOFSEM*, pages 419–427, 2010.
44. T. Gagie, G. Navarro, and S. J. Puglisi. Colored range queries and document retrieval. In *Proc. 17th SPIRE*, pages 67–81, 2010.
45. T. Gagie, G. Navarro, and S. J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theor. Comp. Sci.*, 426-427:25–41, 2012.
46. T. Gagie, S. J. Puglisi, and A. Turpin. Range quantile queries: Another virtue of wavelet trees. In *Proc. 16th SPIRE*, pages 1–6, 2009.
47. S. Gog. *Compressed Suffix Trees: Design, Construction, and Applications.* PhD thesis, Univ. of Ulm, Germany, 2011.
48. A. Golynski. Optimal lower bounds for rank and select indexes. In *Proc. 33th ICALP*, pages 370–381, 2006.
49. A. Golynski. Optimal lower bounds for rank and select indexes. *Theor. Comp. Sci.*, 387(3):348–359, 2007.
50. A. Golynski, R. Grossi, A. Gupta, R. Raman, and S. S. Rao. On the size of succinct indices. In *Proc. 15th ESA*, pages 371–382, 2007.
51. A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th SODA*, pages 368–373, 2006.
52. G. Gonnet, R. Baeza-Yates, and T. Snider. *Information Retrieval: Data Structures and Algorithms*, chapter 3: New indices for text: Pat trees and Pat arrays, pages 66–82. Prentice-Hall, 1992.

53. R. González and G. Navarro. Compressed text indexes with fast locate. In *Proc. 18th CPM*, pages 216–227, 2007.
54. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th SODA*, pages 841–850, 2003.
55. R. Grossi, A. Gupta, and J. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. In *Proc. 15th SODA*, pages 636–645, 2004.
56. R. Grossi and G. Ottaviano. The wavelet trie: Maintaining an indexed sequence of strings in compressed space. In *Proc. 31st PODS*, 2012. To appear.
57. R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. 32nd STOC*, pages 397–406, 2000.
58. R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comp.*, 35(2):378–407, 2006.
59. R. Grossi, J. Vitter, and B. Xu. Wavelet trees: From theory to practice. In *Proc. 1st CCP*, pages 210–221, 2011.
60. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. 10th SIGMOD*, pages 47–57, 1984.
61. M. He and I. Munro. Succinct representations of dynamic strings. In *Proc. 17th SPIRE*, pages 334–346, 2010.
62. D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the I.R.E.*, 40(9):1090–1101, 1952.
63. G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th FOCS*, pages 549–554, 1989.
64. J. JáJá, C. Mortensen, and Q. Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In *Proc. 15th ISAAC*, pages 558–568, 2004.
65. A. G. Jørgensen and K. D. Larsen. Range selection and median: Tight cell probe lower bounds and adaptive data structures. In *Proc. 22nd SODA*, pages 805–813, 2011.
66. J. Kärkkäinen. *Repetition-Based Text Indexing.* PhD thesis, Univ. of Helsinki, Finland, 1999.
67. J. Kärkkäinen and S. J. Puglisi. Fixed block compression boosting in FM-indexes. In *Proc. 18th SPIRE*, pages 174–184, 2011.
68. S. Kreft and G. Navarro. Self-indexing based on LZ77. In *Proc. 22nd CPM*, pages 41–54, 2011.
69. J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.
70. V. Mäkinen and G. Navarro. New search algorithms and time/space tradeoffs for succinct suffix arrays. Tech. Rep. C-2004-20, Univ. of Helsinki, Finland, April 2004.
71. V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic J. Comp.*, 12(1):40–66, 2005.
72. V. Mäkinen and G. Navarro. Position-restricted substring searching. In *Proc. 7th LATIN*, pages 703–714, 2006.
73. V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *Proc. 14th SPIRE*, pages 214–226, 2007.
74. V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theor. Comp. Sci.*, 387(3):332–347, 2007.
75. V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comp. Biol.*, 17(3):281–308, 2010.

76. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comp.*, 22(5):935–948, 1993.
77. G. Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001.
78. E. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.
79. I. Munro. Tables. In *Proc. 16th FSTTCS*, pages 37–42, 1996.
80. S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. 13th SODA*, pages 657–666, 2002.
81. G. Navarro. Indexing text using the Ziv-Lempel trie. *J. Discr. Alg.*, 2(1):87–114, 2004.
82. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comp. Surv.*, 39(1):article 2, 2007.
83. G. Navarro, Y. Nekrich, and L. Russo. Space-efficient data-analysis queries on grids. *CoRR*, abs/1106.4649v2, 2012.
84. G. Navarro and E. Providel. Fast, small, simple rank/select on bitmaps. In *Proc. 11th SEA*, 2012. To appear.
85. G. Navarro and S. J. Puglisi. Dual-sorted inverted lists. In *Proc. 17th SPIRE*, pages 310–322, 2010.
86. G. Navarro, S. J. Puglisi, and D. Valenzuela. Practical compressed document retrieval. In *Proc. 10th SEA*, pages 193–205, 2011.
87. G. Navarro and L. Russo. Space-efficient data-analysis queries on grids. In *Proc. 22nd ISAAC*, pages 323–332, 2011.
88. G. Navarro and K. Sadakane. Fully-functional static and dynamic succinct trees. *CoRR*, abs/0905.0768v5, 2010.
89. D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th ALENEX*, 2007.
90. M. Pătraşcu. Lower bounds for 2-dimensional range counting. In *Proc. 39th STOC*, pages 40–46, 2007.
91. M. Pătraşcu. Succincter. In *Proc. 49th FOCS*, pages 305–313, 2008.
92. M. Pătraşcu and E. Viola. Cell-probe lower bounds for succinct partial sums. In *Proc. 21st SODA*, pages 117–122, 2010.
93. R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th SODA*, pages 233–242, 2002.
94. T. Schnattinger, E. Ohlebusch, and S. Gog. Bidirectional search in a string with wavelet trees. In *Proc. 21st CPM*, pages 40–50, 2010.
95. J. Sirén, N. Välimäki, V. Mäkinen, and G. Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *Proc. 15th SPIRE*, pages 164–175, 2008.
96. G. Tischler. On wavelet tree construction. In *Proc. 22nd CPM*, pages 208–218, 2011.
97. N. Välimäki and V. Mäkinen. Space-efficient algorithms for document retrieval. In *Proc. 18th CPM*, pages 205–215, 2007.
98. P. Weiner. Linear pattern matching algorithm. In *Proc. 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.
99. I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes.* Morgan Kaufmann, 2nd edition, 1999.
100. C.-C. Yu, W.-K. Hon, and B.-F. Wang. Efficient data structures for the orthogonal range successor problem. In *Proc. 15th COCOON*, pages 96–105, 2009.