The Wavelet Matrix: An Efficient Wavelet Tree for Large Alphabets [*]

Francisco Claude[†]
Esc. Inf. & Tel.
Univ. Diego Portales, Chile
fclaude@recoded.cl

Gonzalo Navarro[‡]
Dept. of Computer Science
Univ. of Chile, Chile
gnavarro@dcc.uchile.cl

Alberto Ordóñez[§]
Database Laboratory
Univ. da Coruña, Spain
alberto.ordonez@udc.es

Abstract

The *wavelet tree* is a flexible data structure that permits representing sequences $S[1,n]$ of symbols over an alphabet of size σ, within compressed space and supporting a wide range of operations on S. When σ is significant compared to n, current wavelet tree representations incur in noticeable space or time overheads. In this article we introduce the *wavelet matrix*, an alternative representation for large alphabets that retains all the properties of wavelet trees but is significantly faster. We also show how the wavelet matrix can be compressed up to the zero-order entropy of the sequence without sacrificing, and actually improving, its time performance. Our experimental results show that the wavelet matrix outperforms all the wavelet tree variants along the space/time tradeoff map.

1 Introduction

In many applications related to text indexing and succinct data structures, it is necessary to represent a sequence $S[1,n]$ over an integer alphabet $[0,\sigma)$ so as to support the following functionality:

- $\texttt{access}(S,i)$ returns $S[i]$.
- $\texttt{rank}_a(S,i)$ returns the number of occurrences of symbol a in $S[1,i]$.
- $\texttt{select}_a(S,j)$ returns the position in S of the j-th occurrence of symbol a.

Some examples where this problem arises are indexes for supporting indexed pattern matching on strings [35, 36, 28, 29, 52], indexes for solving computational biology problems on sequences [63, 11], simulation of inverted indexes over natural language text collections [14, 2], representation

[*]An early partial versions of this article appeared in *Proc. SPIRE 2012* [23].

[†]Funded in part by CONICYT Fondecyt Iniciación 11130104.

[‡]Funded in part by Millennium Nucleus Information and Coordination in Networks ICM/FIC P10-024F, Chile.

[§]Funded by CDTI EXP 000645663/ITC-20133062 (CDTI, Ministerio de Economía y Competitividad-MEC-, and Axencia Galega de Innovación -AGI-), Xunta de Galicia (Partially by FEDER) ref. GRC2013/053, and by MICINN (PGE and FEDER) refs. TIN2009-14560-C03-02, TIN2010-21246-C02-01, andAP2010-6038 (FPU Program).

of labeled trees and XML structures [12, 3, 27, 1, 8], representation of binary relations and graphs [7, 21, 5, 8], solving document retrieval problems [66, 31], and many more.

An elegant data structure to solve this problem is the *wavelet tree* [35]. In its most basic form, this is a balanced tree of $O(\sigma)$ nodes storing bitmaps. It requires $n \lg \sigma + o(n \lg \sigma) + O(\sigma \lg n)$ bits to represent S and solves the three queries in time $O(\lg \sigma)$. The wavelet tree supports not only the three queries we have mentioned, but more general range search operations that find applications in representing geometric grids [17, 13, 15, 5, 55] and text indexes based on them [50, 28, 45, 18, 22, 42, 53], complex queries on numeric sequences [32, 42, 30], and many others. Various recent surveys [52, 26, 37, 47, 51] are dedicated, partially or totally, to the number of applications of this versatile data structure.

In various applications, the alphabet size σ is significant compared to the length n of the sequence. Some examples are sequences of words (seen as integer tokens) when simulating inverted indexes, sequences of XML tags, and sequences of document numbers in document retrieval. When using wavelet trees to represent grids, the sequence length n becomes the width of the grid and the alphabet size becomes the height of the grid, and both are equal in most cases. Finally, the problem arises when a sequence is indexed as many short sequences, each of which is relatively short compared to the alphabet [41].

A large value of σ affects the space usage of wavelet trees. A pointerless wavelet tree [45] concatenates all the bitmaps levelwise and removes the $O(\sigma \lg n)$ bits from the space. It retains the time complexity of pointer-based wavelet trees, albeit it is slower in practice. This representation can be made to use $nH_0(S) + o(n \lg \sigma)$ bits, where $H_0(S) \leq \lg \sigma$ is the per-symbol zero-order entropy of S, by using compressed bitmaps [61, 35]. This makes the wavelet tree traversal even slower in practice, however.

A pointer-based wavelet tree, instead, can achieve zero-order compression by replacing the balanced tree by the Huffman tree [39]. Then, even without compressing the bitmaps, the storage space becomes $n(H_0(S) + 1) + o(n(H_0(S) + 1)) + O(\sigma \lg n)$ bits. Adding bitmap compression removes the n bits of the Huffman redundancy. In addition, this technique is faster than the basic one, as the average access time is $O(H_0(S))$. By using canonical Huffman codes [64], a pointerless Huffman-shaped wavelet tree is also possible [68]. This removes the $O(\sigma \lg n)$ bits of the pointers, whereas those of the Huffman model can be reduced to $\sigma \lg \sigma + O(\sigma)$, and even $O(\sigma \lg \lg n)$, in the case of a canonical code [56]. Therefore the total space can be written as $nH_0(S) + o(n(H_0(S) + 1)) + O(\sigma \lg \lg n)$ bits.

Other than wavelet trees, Golynski et al. [33] proposed a sequence representation for large alphabets, which uses $n \lg \sigma + o(n \lg \sigma)$ bits (no compression) and offers much faster time complexities to support the three operations, $O(\lg \lg \sigma)$. Later, Barbay et al. [6] built on this idea to obtain zero-order compression, $nH_0(S) + o(n(H_0(S) + 1))$ bits, while retaining the times. This so-called "alphabet-partitioned" representation does not, however, offer the richer functionality of wavelet trees. Moreover, as shown in their experiments [4], its sublinear space terms are higher in practice than those of a zero-order compressed wavelet tree (yet their better complexity does show up in practice). There are recent theoretical developments slightly improving those complexities [10], but their sublinear space terms would be even higher in practice.

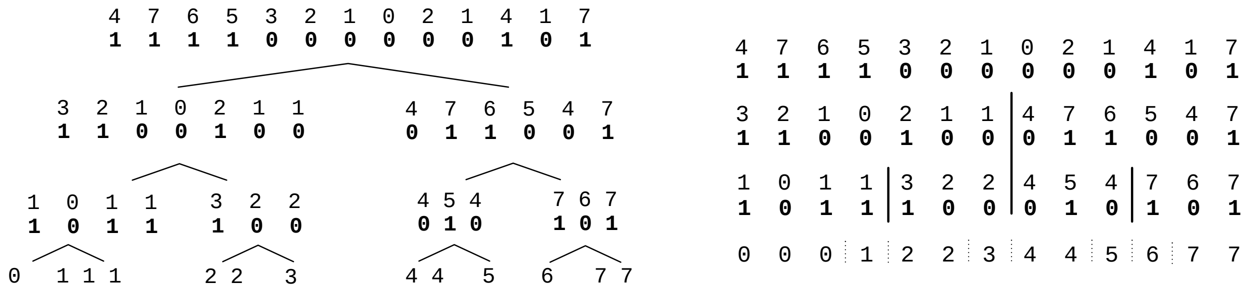


Figure 1: On the left, the standard wavelet tree over a sequence. The subsequences S_v are not stored. The bitmaps B_v, in bold, are stored, as well as the tree topology. On the right, its pointerless version. The divisions into nodes are not stored but computed on the fly.

Our contribution. In this article we introduce the *wavelet matrix*. This is an alternative representation of the balanced pointerless wavelet tree that reorders the nodes in each level, in a way that retains all the wavelet tree functionality while the traversals needed to carry out the operations are simplified and sped up. The wavelet matrix then retains all the capabilities of wavelet trees, is resistant to large alphabets, and its speed gets close to that of pointer-based wavelet trees. It can also obtain zero-order compression by compressing the bitmaps (which slows it down).

Then, we focus on combining wavelet matrices with Huffman-shaped wavelet trees, so as to obtain Huffman-shaped wavelet matrices. These yield simultaneously zero-order compression and fast operations. It turns out, however, that the canonical Huffman codes cannot be directly combined with the node numbering induced by the wavelet matrix, so we derive an alternative code assignment scheme that is also optimal and compatible with the wavelet matrix.

We implement all the variants and test them over various real-life sequences, showing that a few versions of the wavelet matrix dominate all the wavelet tree variants across the space/time tradeoff map, on diverse sequences over large alphabets and point grids.

2 Basic Concepts

2.1 Wavelet Trees

A wavelet tree [35] for sequence $S[1,n]$ over alphabet $[0..\sigma)$ is a complete balanced binary tree, where each node handles a range of symbols. The root handles $[0..\sigma)$ and each leaf handles one symbol. Each node v handling the range $[\alpha_v, \omega_v)$ represents the subsequence $S_v[1,n_v]$ of S formed by the symbols in $[\alpha_v, \omega_v)$, but it does not explicitly store S_v. Rather, internal nodes v store a bitmap $B_v[1,n_v]$, so that $B_v[i] = 0$ if $S_v[i] < \alpha_v + 2^{\lceil \lg(\omega_v - \alpha_v) \rceil - 1}$ and $B_v[i] = 1$ otherwise. That is, we partition the alphabet interval $[\alpha_v, \omega_v)$ into two roughly equal parts: a "left" one, $[\alpha_v, \alpha_v + 2^{\lceil \lg(\omega_v - \alpha_v) \rceil - 1})$ and a "right" one, $[\alpha_v + 2^{\lceil \lg(\omega_v - \alpha_v) \rceil - 1}, \omega_v)$. These are handled by the left and right children of v. No bitmaps are stored for the leaves. Figure 1 (left) gives an example.

The tree has height $\lceil \lg \sigma \rceil$, and it has exactly σ leaves and $\sigma - 1$ internal nodes. If we regard it level by level, we can see that it holds, in the B_v bitmaps, up to n bits per level (all the leaves in each level appear to the right of internal nodes). Thus it stores at most $n\lceil \lg \sigma \rceil$ bits. Storing the

Algorithm 1 Standard wavelet tree algorithms: On the wavelet tree of sequence S rooted at ν, $\mathbf{acc}(\nu, i)$ returns $S[i]$; $\mathbf{rnk}(\nu, a, i)$ returns $\mathtt{rank}_a(S, i)$; and $\mathbf{sel}(\nu, a, j)$ returns $\mathtt{select}_a(S, j)$. The left/right children of v are called v_l/v_r.

```
acc(v, i)
    if $\omega_v - \alpha_v = 1$ then
        return $\alpha_v$
    end if
    if $B_v[i] = 0$ then
        $i \leftarrow \mathtt{rank}_0(B_v, i)$
        return acc($v_l, i$)
    else
        $i \leftarrow \mathtt{rank}_1(B_v, i)$
        return acc($v_r, i$)
    end if

rnk(v, a, i)
    if $\omega_v - \alpha_v = 1$ then
        return $i$
    end if
    if $a < 2^{\lceil \lg(\omega_v - \alpha_v) \rceil - 1}$ then
        $i \leftarrow \mathtt{rank}_0(B_v, i)$
        return rnk($v_l, a, i$)
    else
        $i \leftarrow \mathtt{rank}_1(B_v, i)$
        return rnk($v_r, a, i$)
    end if

sel(v, a, j)
    if $\omega_v - \alpha_v = 1$ then
        return $j$
    end if
    if $a < 2^{\lceil \lg(\omega_v - \alpha_v) \rceil - 1}$ then
        $j \leftarrow$ sel($v_l, a, j$)
        return $\mathtt{select}_0(B_v, j)$
    else
        $j \leftarrow$ sel($v_r, a, j$)
        return $\mathtt{select}_1(B_v, j)$
    end if
```

tree pointers, and pointers to the bitmaps, requires $O(\sigma \lg n)$ further bits, if we use the minimum of $\lg n$ bits for the pointers.

To access $S[i]$, we start from the root node ν, setting $i_\nu = i$. If $B_\nu[i_\nu] = 0$, this means that $S[i] = S_\nu[i_\nu] < 2^{\lceil \lg \sigma \rceil - 1}$ and that the symbol is represented in the subsequence S_{ν_l} of the left child ν_l of the root. Otherwise, $S_\nu[i_\nu] \geq 2^{\lceil \lg \sigma \rceil - 1}$ and it is represented in the subsequence S_{ν_r} of the right child ν_r of the root. In the first case, the position of $S_\nu[i_\nu]$ in S_{ν_l} is $i_{\nu_l} = \mathtt{rank}_0(B_\nu, i_\nu)$, whereas in the second, the position in S_{ν_r} is $i_{\nu_r} = \mathtt{rank}_1(B_\nu, i_\nu)$. We continue recursively, extracting $S_v[i_v]$ from node $v = \nu_l$ or $v = \nu_r$, until we arrive at a leaf representing the alphabet interval $[a, a]$, where we can finally report $S[i] = a$.

Therefore, the maximum cost of operation `access` is that of $\lceil \lg \sigma \rceil$ binary `rank` operations on bitmaps B_v. Binary `rank` and `select` operations can be carried out in constant time using only $o(n_v)$ bits on top of B_v [40, 49, 19].

The process to compute $\mathtt{rank}_a(S, i)$ is similar. The difference is that we do not descend according to whether $B_v[i]$ equals 0 or 1, but rather according to the bits of $a \in [0, \sigma)$: the highest bit of a tells us whether to go left or right, and the lower bits are used in the next levels. When moving to a child u of v, we compute $i_u = \mathtt{rank}_{0/1}(B_v, i_v)$ to be the number of times the current bit of a appears in $B_v[1, i_v]$. When we arrive at the leaf u handling the range $[a, a]$, the answer to $\mathtt{rank}_a(S, i)$ is i_u.

Finally, to compute $\mathtt{select}_a(S, j)$ we must proceed upwards. We start at the leaf u that handles the alphabet range $[a, a]$. So we want to track the position of $S_u[j_u]$, $j_u = j$, towards the root. If u is the left child of its parent v, then the corresponding position at the parent is $S_v[j_v]$, where $j_v = \mathtt{select}_0(B_v, j_u)$. Else, the corresponding position is $j_v = \mathtt{select}_1(B_v, j_u)$. When we finally arrive at the root ν, the answer to the query is j_ν.

Thus the maximum cost of query $\mathtt{rank}_a(S, i)$ is $\lceil \lg \sigma \rceil$ binary `rank` operations (just like $\mathtt{access}(S, i)$), and the maximum cost of query $\mathtt{select}_a(S, i)$ is $\lceil \lg \sigma \rceil$ binary `select` operations. Algorithm 1 gives the pseudocode (the recursive form is cleaner, but recursion can be easily removed).

2.2 Pointerless Wavelet Trees

Since the internal nodes at each level of the wavelet tree are all to the left, it is possible to concatenate all the bitmaps at each level and still retain the same functionality [45]. Instead of a bitmap per node v, there will be a single bitmap per level ℓ, $\tilde{B}_\ell[1,n]$. Figure 1 (right) illustrates this arrangement. The main problem is how to keep track of the range $\tilde{B}_\ell[s_v, e_v]$ corresponding to a node v of depth ℓ.

The strict variant. The strict variant [45] stores no data apart from the $\lceil \lg \sigma \rceil$ pointers to the level bitmaps. Keeping track of the node ranges is not hard if we start at the root (as in `access` and `rank`). Initially, we know that $[s_\nu, e_\nu] = [1, n]$, that is, the whole bitmap $\tilde{B}_0$ is equal to the bitmap of the root, B_ν. Now, imagine that we have navigated towards a node v at depth ℓ, and know $[s_v, e_v]$. The two children of v share the same interval $[s_v, e_v]$ at $\tilde{B}_{\ell+1}$. The split point is $m = \texttt{rank}_0(\tilde{B}_\ell, e_v) - \texttt{rank}_0(\tilde{B}_\ell, s_v - 1)$, the number of 0s in $\tilde{B}_\ell[s_v, e_v]$. Then, if we descend to the left child v_l, we will have $[s_{v_l}, e_{v_l}] = [s_v, s_v + m - 1]$. If we descend to the right child v_r, we will have $[s_{v_r}, e_{v_r}] = [s_v + m, e_v]$.

Things are a little bit harder for `select`, because we must proceed upwards. In the strict variant, the way to carry out $\texttt{select}_a(S, j)$ is to first descend to the leaf corresponding to symbol a, and then track the leaf position j up to the root as we return from the recursion.

Algorithm 2 gives the pseudocode (we use $p = s - 1$ instead of $s = s_v$). Note that, compared to the standard version, the strict variant requires two extra binary `rank` operations per original binary `rank`, on the top-down traversals (i.e., for queries `access` and `rank`). For query `select`, the strict variant requires two extra binary `rank` operations per original binary `select`. Thus the times may up to triple for these traversals.[1]

The extended variant. The *extended* variant [20], instead, stores an array $C[0, \sigma - 1]$ of pointers to the σ starting positions of the symbols in the (virtual) array of the leaves, or said another way, $C[a]$ is the number of occurrences of symbols smaller than a in S. Note this array requires $O(\sigma \lg n)$ bits (or at best $O(\sigma \lg(n/\sigma)) + o(n)$ if represented as a compressed bitmap [61]), but the constant is much lower than on a pointer-based tree (which stores the left child, the right child, the parent, the value n_v, the pointer to bitmap B_v, pointers to the leaves, etc.). We also need to indicate the level in which each leaf is found.

With the help of array C, the number of operations becomes closer to the standard version, since C lets us compute the ranges: The range of any node v is simply $[s_v, e_v] = [C[\alpha_v] + 1, C[\omega_v]]$. In the algorithms for queries `access` and `rank`, where we descend from the root, the values α_v and ω_v are easily maintained. Thus we do not need to compute r in Algorithm 2, as it is used only to compute $e = e_v = C[\omega_v]$. Thus we require only one extra binary `rank` operation per level.

This is slightly more complicated when solving query $\texttt{select}_a(S, j)$. We start at offset j in the interval $[C[\alpha_u] + 1, C[\omega_u]]$ for $(\alpha_u, \omega_u) = (a, a + 1)$ and track this position upwards: If the leaf u is a left child of its parent v (i.e., if α_u is even), then the parent's range (in the deepest bitmap $\tilde{B}_\ell$) is $(\alpha_v, \omega_v) = (\alpha_u, \omega_u + 1)$. Instead, if the leaf is a right child of its parent, then the parent's range is

[1]In practice the effect is not so large because of cache effects when s_v is close to e_v. In addition, binary `select` is more expensive than `rank` in practice, thus the impact on query `select` is lower.

Algorithm 2 Pointerless wavelet tree algorithms (strict variant): On the wavelet tree of sequence S, $\mathbf{acc}(0,i,0,n)$ returns $S[i]$; $\mathbf{rnk}(0,a,i,0,n)$ returns $\mathtt{rank}_a(S,i)$; and $\mathbf{sel}(0,a,j,0,n)$ returns $\mathtt{select}_a(S,j)$. For simplicity we have omitted the computation of $[\alpha_v,\omega_v)$.

```
acc(ℓ, i, p, e)
    if $\omega_v - \alpha_v = 1$ then
        return $\alpha_v$
    end if
    $l \leftarrow \mathtt{rank}_0(\tilde{B}_\ell, p)$
    $r \leftarrow \mathtt{rank}_0(\tilde{B}_\ell, e)$
    if $\tilde{B}_\ell[i] = 0$ then
        $z \leftarrow \mathtt{rank}_0(\tilde{B}_\ell, p+i)$
        return acc($\ell$+1,
            $z-l, p, p+r-l$)
    else
        $z \leftarrow \mathtt{rank}_1(\tilde{B}_\ell, p+i)$
        return acc($\ell$+1,
            $z-(p-l), p+r-l, e$)
    end if
```

```
rnk(ℓ, a, i, p, e)
    if $\omega_v - \alpha_v = 1$ then
        return $i$
    end if
    $l \leftarrow \mathtt{rank}_0(\tilde{B}_\ell, p)$
    $r \leftarrow \mathtt{rank}_0(\tilde{B}_\ell, e)$
    if $a < 2^{\lceil \lg(\omega_v - \alpha_v) \rceil - 1}$ then
        $z \leftarrow \mathtt{rank}_0(\tilde{B}_\ell, p+i)$
        return rnk($\ell$+1, $a$,
            $z-l, p, p+r-l$)
    else
        $z \leftarrow \mathtt{rank}_1(\tilde{B}_\ell, p+i)$
        return rnk($\ell$+1, $a$,
            $z-(p-l), p+r-l, e$)
    end if
```

```
sel(ℓ, a, j, p, e)
    if $\omega_v - \alpha_v = 1$ then
        return $j$
    end if
    $l \leftarrow \mathtt{rank}_0(\tilde{B}_\ell, p)$
    $r \leftarrow \mathtt{rank}_0(\tilde{B}_\ell, e)$
    if $a < 2^{\lceil \lg(\omega_v - \alpha_v) \rceil - 1}$ then
        $j \leftarrow$ sel($\ell$+1, $a, j, p, p+r-l$)
        return $\mathtt{select}_0(\tilde{B}_\ell, l+j) - p$
    else
        $j \leftarrow$ sel($\ell$+1, $a, j, p+r-l, e$)
        return $\mathtt{select}_1(\tilde{B}_\ell, (p-l)+j) - p$
    end if
```

$(\alpha_v, \omega_v) = (\alpha_u - 1, \omega_u)$. We use binary `select` on the range $[C[\alpha_v]+1, C[\omega_v]]$ to map the position j to the parent's range. Now we proceed similarly at the parent w of v. If $\alpha_v = 0 \bmod 4$, then v is the left child of w, otherwise it is the right child. In the first case, the range of w in bitmap $\tilde{B}_{\ell-1}$ is $(\alpha_w, \omega_w) = (\alpha_v, \omega_v + 2)$, otherwise it is $(\alpha_w, \omega_w) = (\alpha_v - 2, \omega_v)$. We continue until the root, where j is the answer. In this case we need only one extra binary `rank` operation per level. Algorithm 3 details the algorithms.

2.3 Huffman Shaped Wavelet Trees

Given the frequencies of the σ symbols in $S[1,n]$, the Huffman algorithm [39] produces an optimal variable-length encoding so that (1) it is prefix-free, that is, no code is a prefix of another; (2) the size of the compressed sequence is minimized. If symbol $a \in [0,\sigma)$ appears n_a times in S, then the Huffman algorithm will assign it a codeword of length ℓ_a so that the sum $L = \sum_a n_a \ell_a$ is minimized. Then the file is compressed to L bits by replacing each symbol $S[i] = a$ by its code of length ℓ_a. The *empirical zero-order entropy* [25] of S is $H_0(S) = \sum_a \frac{n_a}{n} \lg \frac{n}{n_a} \le \lg \sigma$, and no statistical compressor based on individual symbol probabilities can output less than $nH_0(S)$ bits. The output size of Huffman compression can be bounded by $\sum_a n_a \ell_a < n(H_0(S)+1)$ bits, which is off the optimum by less than 1 bit per symbol.

Huffman [39] showed how to build a so-called *Huffman tree* to obtain these codes. The tree leaves will contain the symbols, whose codes are obtained by following the tree path from the root to their leaves. Each branch of the tree is labeled with 0 (say, the left child) or 1 (say, the right child), and the code associated with a symbol a is obtained by concatenating the labels found in the path from the tree root to the leaf that contains symbol a.

Building a balanced wavelet tree is equivalent to using a fixed-length encoding. Instead, by

Algorithm 3 Pointerless wavelet tree algorithms (extended variant): On the wavelet tree of sequence S, $\mathbf{acc}(0,i)$ returns $S[i]$; $\mathbf{rnk}(0,a,i)$ returns $\texttt{rank}_a(S,i)$; and $\mathbf{sel}(a,j)$ returns $\texttt{select}_a(S,j)$. For simplicity we have omitted the computation of $[\alpha_v,\omega_v)$, except on $\mathbf{sel}(a,j)$, where for simplicity we assume $C[a]$ refers to level $\ell = \lceil \lg \sigma \rceil$.

```
acc(ℓ, i)
   if $\omega_v - \alpha_v = 1$ then
      return $\alpha_v$
   end if
   $l \leftarrow \texttt{rank}_0(\tilde{B}_\ell, C[\alpha_v])$
   $z \leftarrow \texttt{rank}_0(\tilde{B}_\ell, C[\alpha_v]+i)$
   if $\tilde{B}_\ell[i] = 0$ then
      return acc($\ell+1, z-l$)
   else
      return acc($\ell+1, i-(z-l)$)
   end if
```

```
rnk(ℓ, a, i)
   if $\omega_v - \alpha_v = 1$ then
      return $i$
   end if
   $l \leftarrow \texttt{rank}_0(\tilde{B}_\ell, C[\alpha_v])$
   $z \leftarrow \texttt{rank}_0(\tilde{B}_\ell, C[\alpha_v] + i)$
   if $a < 2^{\lceil \lg(\omega_v - \alpha_v) \rceil - 1}$ then
      return rnk($\ell+1, a, z-l$)
   else
      return rnk($\ell+1, a, i-(z-l)$)
   end if
```

```
sel(a, j)
   $\ell \leftarrow \lceil \lg \sigma \rceil$, $d \leftarrow \lceil \lg \sigma \rceil - \ell + 1$
   while $\ell \geq 0$ do
      if $a \bmod 2^d = 0$ then
         $l \leftarrow \texttt{rank}_0(\tilde{B}_\ell, C[\alpha_v])$
         $j \leftarrow \texttt{select}_0(\tilde{B}_\ell, l+j)$
      else
         $\alpha_v \leftarrow \alpha_v - 2^{d-1}$
         $l \leftarrow \texttt{rank}_1(\tilde{B}_\ell, C[\alpha_v])$
         $j \leftarrow \texttt{select}_1(\tilde{B}_\ell, l+j)$
      end if
      $j \leftarrow j - C[\alpha_v]$
      $\ell \leftarrow \ell - 1$, $d \leftarrow d + 1$
   end while
   return $j$
```

giving the wavelet tree the shape of the Huffman tree, the total number of bits stored is exactly the output size of the Huffman compressor [35, 51]: The leaf of a is at depth ℓ_a, and each of the n_a occurrences induces one bit in the bitmap of each of the ℓ_a ancestors of the leaf. The size of this tree, plus `rank`/`select` overheads, is thus upper bounded by $n(H_0(S)+1) + o(n(H_0(S)+1)) + O(\sigma \lg n)$ bits. Figure 2 depicts a Huffman-shaped wavelet tree.

The wavelet tree operations are performed verbatim on Huffman-shaped wavelet trees. Moreover, they become faster on average: If $i \in [1,n]$ is chosen at random for $\texttt{access}(S,i)$, or a is chosen with probability n_a/n in operations $\texttt{rank}_a(S,i)$ and $\texttt{select}_a(S,j)$ (which is the typical case in most applications), then the average time is $O(H_0(S)+1)$. By rebalancing deep leaves, the space and average time are maintained and the worst-case time of the operations is limited to $O(\lg \sigma)$ [9].

Zero-order compression can also be achieved on the balanced wavelet tree, by using a compressed representation of the bitmaps [61]. The time remains the same and the space decreases to $nH_0(S) + o(n \lg \sigma)$ bits [35]. Combining the compressed bitmap representation with Huffman shape, we obtain $nH_0(S) + o(n(H_0(S)+1)) + O(\sigma \lg n)$ bits. This combination works well in practice [20], although the compressed bitmap representation is in practice slower than the plain one.

2.4 Wavelet Trees on Point Grids

As mentioned in the Introduction, wavelet trees are not only useful to support `access`, `rank` and `select` operations on sequences. They are also frequently used to represent point grids [17, 51], where they can for example count or list the points that lie in a rectangular area. Typically the grid is square, of $n \times n$ cells, and contains n points, exactly one point per row and per column (other arrangements are routinely mapped to this simplified case). Then it can be regarded as a sequence

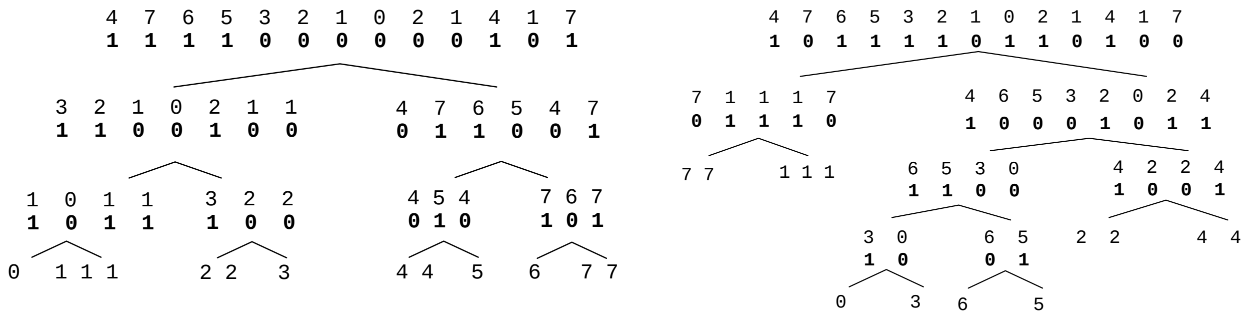


Figure 2: On the left, the same wavelet tree of Figure 1. On the right, its Huffman-shaped version.

$S[1, n]$ over a large alphabet of size $\sigma = n$. In this case, pointer-based wavelet trees perform poorly, as the space for the pointers is dominant. Similarly, zero-order compression is ineffective. The balanced wavelet trees without pointers [45] are the most successful representation.

The pseudocode for range searches using standard wavelet trees is easily available, see for example Gagie et al. [31]. Algorithm 4 shows the algorithms adapted to pointerless wavelet trees. We consider the two basic operations `count`(P, x_1, x_2, y_1, y_2) and `report`(P, x_1, x_2, y_1, y_2), which count and list, respectively, the points within the rectangle $[x_1, x_2] \times [y_1, y_2]$ from the point set represented in sequence $P[1, n]$. The time complexities can be shown to be $O(\lg n)$ for `count` and $O(k \lg(n/k))$ for a `report` operation that lists k points. In practical terms, compared to the standard versions, the pointerless algorithms requires twice the number of `rank` operations.

3 Pointerless Huffman Shaped Wavelet Trees

In this section we revisit the technique [68] to use canonical Huffman codes [64] to represent Huffman-shaped wavelet trees without pointers, this way removing the main component of the $O(\sigma \lg n)$ extra bits and retaining the advantages of reduced space and $O(H_0(S) + 1)$ average traversal time.

The problem that arises when storing a standard Huffman-shaped wavelet tree in levelwise form is that a leaf that appears in the middle of a level leaves a "hole" that ruins the calculations done at the nodes to the right of it to find their position in the next level. Canonical Huffman codes choose one of the many optimal Huffman trees that, among other interesting benefits [64, 62], yields a set of codes in which longer codes appear to the left of shorter codes.[2] As a consequence, all the leaves of a level appear grouped to the right, and therefore do not alter the navigation calculations for the other nodes. The levelwise deployment of the tree can be seen as a sequence of "contiguous" bitmaps of varying length.

The navigation procedures of Algorithm 2 can then be used verbatim, except for a few alphabet mappings that must be carried out: For `access`(S, i), we need to maintain the Huffman tree so that, given the 0/1 labels of the traversed path, we determine the alphabet symbol corresponding to that leaf of the Huffman tree. For `rank`$_a(S, i)$, we need to convert the symbol a to its variable-length

[2]It is usually to the right, but this way is more convenient for us.

Algorithm 4 Range search algorithms on pointerless wavelet trees: $\textbf{count}(0, x_1, x_2, y_1, y_2, 0, n)$ returns $\texttt{count}(P, x_1, x_2, y_1, y_2)$ on the wavelet tree of sequence P; and $\textbf{report}(0, x_1, x_2, y_1, y_2, 0, n)$ outputs all those y, where a point with coordinate $y_1 \leq y \leq y_2$ appears in $P[x_1, x_2]$. For simplicity we have omitted the computation of $[\alpha_v, \omega_v)$.

```
count($\ell, x_1, x_2, y_1, y_2, p, e$)
  if $x_1 > x_2 \ \vee\ [\alpha_v, \omega_v] \cap [y_1, y_2] = \emptyset$ then
    return 0
  else if $[\alpha_v, \omega_v] \subseteq [y_1, y_2]$ then
    return $x_2 - x_1 + 1$
  else
    $l \leftarrow \texttt{rank}_0(\tilde{B}_\ell, p)$
    $r \leftarrow \texttt{rank}_0(\tilde{B}_\ell, e)$
    $x_1^l \leftarrow \texttt{rank}_0(\tilde{B}_\ell, x_1 - 1) - l + 1$
    $x_2^l \leftarrow \texttt{rank}_0(\tilde{B}_\ell, x_2) - l$
    $x_1^r \leftarrow x_1 - x_1^l + 1$, $x_2^r \leftarrow x_2 - x_2^l$
    return count($\ell{+}1, x_1^l, x_2^l, y_1, y_2, p, p{+}r{-}l$)
           +count($\ell{+}1, x_1^r, x_2^r, y_1, y_2, p{+}r{-}l, e$)
  end if
```

```
report($v, x_1, x_2, y_1, y_2, p, e$)
  if $x_1 > x_2 \ \vee\ [\alpha_v, \omega_v] \cap [y_1, y_2] = \emptyset$ then
    return
  else if $\omega_v - \alpha_v = 1$ then
    output $\alpha_v$
  else
    $l \leftarrow \texttt{rank}_0(\tilde{B}_\ell, p)$
    $r \leftarrow \texttt{rank}_0(\tilde{B}_\ell, e)$
    $x_1^l \leftarrow \texttt{rank}_0(\tilde{B}_\ell, x_1 - 1) - l + 1$
    $x_2^l \leftarrow \texttt{rank}_0(\tilde{B}_\ell, x_2) - l$
    $x_1^r \leftarrow x_1 - x_1^l + 1$, $x_2^r \leftarrow x_2 - x_2^l$
    report($\ell{+}1, x_1^l, x_2^l, y_1, y_2, p, p{+}r{-}l$)
    report($\ell{+}1, x_1^r, x_2^r, y_1, y_2, p{+}r{-}l, e$)
  end if
```

code, in order to follow the corresponding path in the wavelet tree. Finally, for $\texttt{select}_a(S, i)$, we need the same as for `rank` for the strict variant, or a pointer to the corresponding leaf area in some bitmap $\tilde{B}_\ell$, for the extended variant. The mappings are also used to determine when to stop a top-down traversal. The mapping information amounts to $O(\sigma \lg n)$ bits as well, but it is much less in practice than what is stored for pointer-based wavelet trees, as explained. Moreover, in the case of canonical codes, $\sigma \lg \sigma + O(\sigma)$ bits are sufficient to represent the mappings. It has also been shown that they can be represented within $O(\sigma \lg \lg n)$ bits as well [56].

The maximum number of levels in a Huffman tree is $O(\lg n)$, and as explained it can be made $O(\lg \sigma)$ without affecting the asymptotic performance. Thus the pointers to the levels add up to a negligible $O(\lg^2 n)$ bits. The rest of the space is as for standard Huffman-shaped wavelet trees: $n(H_0(S) + 1) + o(n(H_0(S) + 1))$ bits. Moreover, by using compressed bitmaps [61], the space is reduced to $nH_0(S) + o(n(H_0(S) + 1))$ bits, albeit in practice the navigation is slowed down.

The algorithm to compute a canonical Huffman code [64] starts from the code length assignments ℓ_a produced by the standard Huffman algorithm, and produces a particular Huffman tree with the same code lengths. First, it computes ℓ_{min} and ℓ_{max}, the minimum and maximum code lengths, and array $nCodes[\ell_{min}, \ell_{max}]$, where $nCodes[\ell]$ is the number of codes of length ℓ. Then, the algorithm assigns the codes as follows:

1. $fst[\ell_{min}] = 0^{\ell_{min}}$ (i.e., ℓ_{min} 0s) is the first code of length ℓ_{min}.

2. All the codes of a given length ℓ are consecutive numbers, from $fst[\ell]$ to $last[\ell] = fst[\ell] + nCodes[\ell] - 1$.

3. The first code of the next length $\ell' > \ell$ that has $nCodes[\ell'] > 0$ is $fst[\ell'] = 2^{\ell' - \ell}(last[\ell] + 1)$.

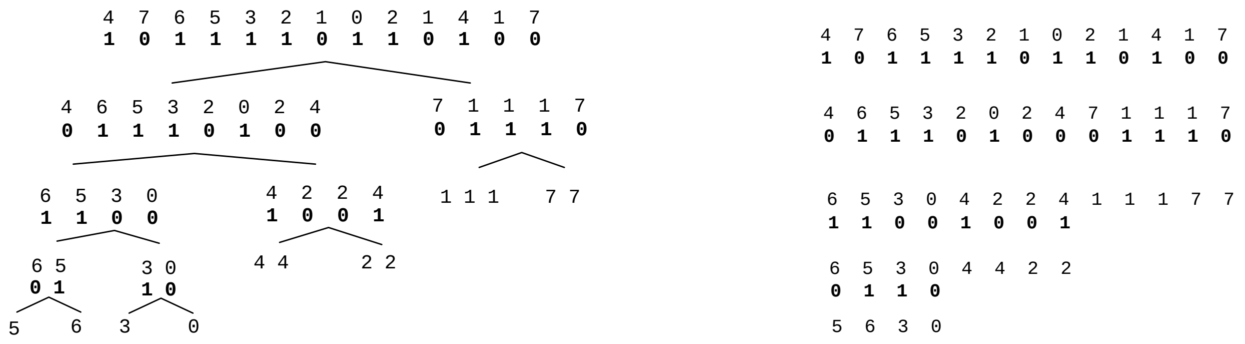


Figure 3: On the left, the pointer-based canonical Huffman code for our running example. On the right, its levelwise representation. Note that from now on we interpret the bit 0 as going right and the bit 1 as going left.

Note that rule 2 ensures that all codes of a given level are consecutive numbers and the first of their length, whereas rule 3 guarantees that the set of produced codes is prefix-free. By interpreting the bit 0 as the right child and the bit 1 as the left child, we have that all the leaves at any level are the rightmost nodes. Figure 3 illustrates the standard and the levelwise deployment of a canonical Huffman code.

4 The Wavelet Matrix

The idea of the wavelet matrix is to break the assumption that the children of a node v, at interval $\tilde{B}_\ell[s_v, e_v]$, must be aligned to it and occupy the interval $\tilde{B}_{\ell+1}[s_v, e_v]$. Freeing the structure from this assumption allows us to design a much simpler mapping mechanism from one level to the next: *all* the zeros of the level go left, and *all* the ones go right. For each level, we will store a single integer z_ℓ that tells the number of 0s in level ℓ. This requires just $O(\lg n \lg \sigma)$ bits, which is insignificant, and allows us to implement the pointerless mechanisms in a simpler and faster way.

More precisely, if $\tilde{B}_\ell[i] = 0$, then the corresponding position at level $\ell + 1$ will be $\mathtt{rank}_0(\tilde{B}_\ell, i)$. If $\tilde{B}_\ell[i] = 1$, the position at level $\ell + 1$ will be $z_\ell + \mathtt{rank}_1(\tilde{B}_\ell, i)$. Note that we can map the position without knowledge of the boundaries of the node the position belongs. Still, every node v at level ℓ occupies a contiguous range in $\tilde{B}_\ell$, as proved next.

Proposition 1. *All the bits in any bitmap $\tilde{B}'_\ell$ of the pointerless wavelet tree that correspond to a wavelet tree node v are also contiguous in the bitmap $\tilde{B}_\ell$ of the the wavelet matrix.*

Proof. This is obviously true for the root $v = \nu$, as it corresponds to the whole $\tilde{B}'_0 = \tilde{B}_0$. Now, assuming it is true for a node v, with interval $\tilde{B}_\ell[s_v, e_v]$, all the positions with $\tilde{B}_\ell[i] = 0$ for $s_v \le i \le e_v$ will be mapped to consecutive positions $\tilde{B}_{\ell+1}[\mathtt{rank}_0(\tilde{B}_\ell, i)]$, and similarly with positions $\tilde{B}_\ell[i] = 1$. □

Figure 4 illustrates the wavelet matrix, where it can be seen that the blocks of the wavelet tree are maintained, albeit in different order. We now describe how to carry out the operations under the strict and the extended variants.

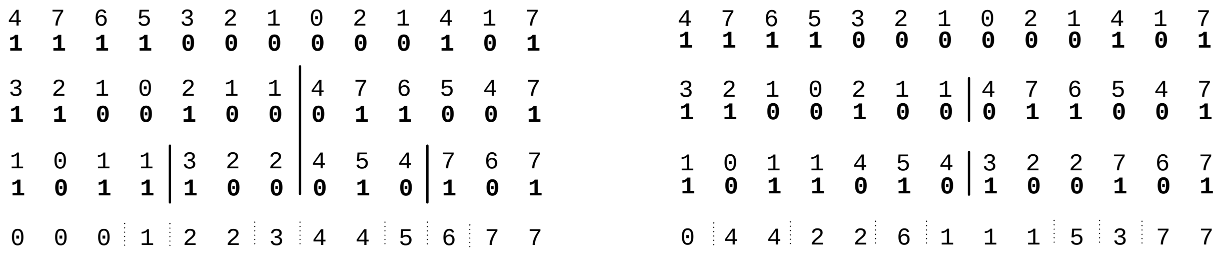


Figure 4: On the left, the pointerless wavelet tree of Figure 1. On the right, the wavelet matrix over the same sequence. One vertical line per level represents the position stored in the z_ℓ values.

The strict variant. To carry out $\texttt{access}(S, i)$, we first set i_0 to i. Then, if $\tilde{B}_0[i_0] = 0$, we set i_1 to $\texttt{rank}_0(\tilde{B}_0, i_0)$. Else we set i_1 to $z_0 + \texttt{rank}_1(\tilde{B}_0, i_0)$. Now we descend to level 1, and continue until reaching a leaf. The sequence of bits $\tilde{B}_\ell[i_\ell]$ read along the way form the value $S[i]$ (or, said another way, we maintain the interval $[\alpha_v, \omega_v)$ and upon reaching the leaf it holds $S[i] = \alpha_v$). Note that we have carried out only one binary `rank` operation per level, just as the standard wavelet tree.

Consider now the computation of $\texttt{rank}_a(S, i)$. This time we need to keep track of the position i, and also of the position preceding the range, initially $p_0 = 0$. At each node v of depth ℓ, if $a < 2^{\lceil \lg(\omega_v - \alpha_v) \rceil - 1}$, then we go "left" by mapping $p_{\ell+1}$ to $\texttt{rank}_0(\tilde{B}_\ell, p_\ell)$ and $i_{\ell+1}$ to $\texttt{rank}_0(\tilde{B}_\ell, i_\ell)$. Otherwise, we go "right" by mapping $p_{\ell+1}$ to $z_\ell + \texttt{rank}_1(\tilde{B}_\ell, p_\ell)$ and $i_{\ell+1}$ to $z_\ell + \texttt{rank}_1(\tilde{B}_\ell, i_\ell)$. When we arrive at the leaf level, the answer is $i_\ell - p_\ell$. Note that we have needed one extra binary `rank` operation per original `rank` operation of the standard wavelet tree, instead of the two extra operations required by the (strict) pointerless variant.

Finally, consider operation $\texttt{select}_a(S, j)$. We first descend towards the leaf of a just as done for $\texttt{rank}_a(S, i)$, keeping track only of p_ℓ. When we arrive at the last level, p_ℓ precedes the range corresponding to the leaf of a, and thus we wish to track upwards position $j_\ell = p_\ell + j$. The upward tracking of a position $\tilde{B}_\ell[j_\ell]$ is simple: If we went left from level $\ell - 1$, then this position was mapped from a 0 in $\tilde{B}_{\ell-1}$, and therefore it came from $j_{\ell-1} = \tilde{B}_{\ell-1}[\texttt{select}_0(\tilde{B}_\ell, j_\ell)]$. Otherwise, position j_ℓ was mapped from a 1, and thus it came from $j_{\ell-1} = \tilde{B}_{\ell-1}[\texttt{select}_1(\tilde{B}_\ell, j_\ell - z_\ell)]$. When we arrive at the root bitmap, j_0 is the answer. Note that we have needed one extra binary `rank` per original binary `select` required by the standard wavelet tree. We remind that in practice `rank` is much less demanding, so this overhead is low. Algorithm 5 gives the pseudocode.

The extended variant. We can speed up `rank` and `select` operations if the array C that points to the starting positions of each symbol in its bitmap is available. First, we note that for $\texttt{rank}_a(S, i)$ we do not need anymore to keep track of p_ℓ, since all we need at the end is to return $i_\ell - C[a]$. Thus the cost becomes similar to that of the standard wavelet tree, which was not achieved with the extended variant of the pointerless wavelet tree.

For $\texttt{select}_a(S, j)$ we can avoid the first downward traversal, as in the pointerless wavelet tree, and use the same technique to determine whether we came from the left or from the right in the parent bitmap. Once again, the cost becomes the same as in a standard wavelet tree, with no extra `rank` operations required. Algorithm 6 gives the detailed algorithm.

Algorithm 5 Wavelet matrix algorithms (strict variant): On the wavelet matrix of sequence S, **acc**$(0, i)$ returns $S[i]$; **rnk**$(0, a, i, 0)$ returns $\mathtt{rank}_a(S, i)$; and **sel**$(0, a, j, 0)$ returns $\mathtt{select}_a(S, j)$. For simplicity we have omitted the computation of $[\alpha_v, \omega_v)$.

```
acc(ℓ, i)
    if $\omega_v - \alpha_v = 1$ then
        return $\alpha_v$
    end if
    if $\tilde{B}_\ell[i] = 0$ then
        $i \leftarrow \mathtt{rank}_0(\tilde{B}_\ell, i)$
    else
        $i \leftarrow \mathtt{rank}_1(\tilde{B}_\ell, i)$
    end if
    return acc($\ell$+1, $i$)
```

```
rnk(ℓ, a, i, p)
    if $\omega_v - \alpha_v = 1$ then
        return $i - p$
    end if
    if $a < 2^{\lceil \lg(\omega_v - \alpha_v) \rceil - 1}$ then
        $p \leftarrow \mathtt{rank}_0(\tilde{B}_\ell, p)$
        $i \leftarrow \mathtt{rank}_0(\tilde{B}_\ell, i)$
    else
        $p \leftarrow z_\ell + \mathtt{rank}_1(\tilde{B}_\ell, p)$
        $i \leftarrow z_\ell + \mathtt{rank}_1(\tilde{B}_\ell, i)$
    end if
    return rnk($\ell$+1, $a, i, p$)
```

```
sel(ℓ, a, j, p)
    if $\omega_v - \alpha_v = 1$ then
        return $p + j$
    end if
    if $a < 2^{\lceil \lg(\omega_v - \alpha_v) \rceil - 1}$ then
        $p \leftarrow \mathtt{rank}_0(\tilde{B}_\ell, p)$
        $j \leftarrow$ sel($\ell$+1, $a, j, p$)
        return $\mathtt{select}_0(\tilde{B}_\ell, j)$
    else
        $p \leftarrow z_\ell + \mathtt{rank}_1(\tilde{B}_\ell, p)$
        $j \leftarrow$ sel($\ell$+1, $a, j, p$)
        return $\mathtt{select}_1(\tilde{B}_\ell, j - z_\ell)$
    end if
```

Range searches. Range searches for rectangles $[x_1, x_2] \times [y_1, y_2]$ require essentially that we are able to track the points x_1 and x_2 downwards in the tree. Thus the same wavelet matrix mechanism for `rank` can be used. Since we are only interested in the value $x_2 - x_1$ at the traversed nodes, we do not need to keep track of p, even in the strict variant (the extended variant requires too much space in this scenario). As a result, we need the same number of `rank` operations as in a pointer-based representation, and get rid of the two extra `rank` operations required by the pointerless wavelet tree. Algorithm 7 gives the pseudocode.

Construction. Construction of the wavelet matrix is simple. At the first level we keep in bitmap $\tilde{B}_0$ the highest bits of the symbols in S, and then stably sort S by those highest bits. Now we keep in bitmap $\tilde{B}_1$ the next-to-highest bits, and stably sort S by those next-to-highest bits. We continue until considering the lowest bit. This takes $O(n \lg \sigma)$ time.

Indeed, we can build the wavelet matrix almost in place, by removing the highest bits after using them and packing the symbols of S. This frees n bits, where we can store the bitmap $\tilde{B}_0$ we have just generated, and keep doing the same for the next levels. We generate the $o(n \lg \sigma)$-space indexes at the end. Thus the construction space is $n\lceil \lg \sigma \rceil + \max(n, o(n \lg \sigma))$ bits. Other more sophisticated techniques [24, 65] may use even less space.

5 The Compressed Wavelet Matrix

Just as on the pointerless wavelet tree, we can achieve zero-order entropy with the wavelet matrix by replacing the plain representations of bitmaps $\tilde{B}_\ell$ by compressed ones [61], the space becoming $nH_0(S) + o(n \lg \sigma)$ bits. Compared to obtaining zero-order entropy using Huffman-shaped trees, this solution has several disadvantages, as explained: (1) the compressed bitmaps are slower to operate than in a plain representation; (2) the number of operations on a Huffman-shaped tree is lower on average than on a balanced tree; (3) the Huffman-shaped wavelet tree is more compact, as

Algorithm 6 Wavelet matrix algorithms (extended variant): On the wavelet matrix of sequence S, $\mathbf{acc}(0,i)$ returns $S[i]$; $\mathbf{rnk}(0,a,i)$ returns $\mathtt{rank}_a(S,i)$; and $\mathbf{sel}(a,j)$ returns $\mathtt{select}_a(S,j)$. For simplicity we have omitted the computation of $[\alpha_v,\omega_v)$, and in $\mathbf{sel}(a,j)$ we assume $C[a]$ refers to level $\ell = \lceil \lg \sigma \rceil$.

```
acc($\ell, i$)
    if $\omega_v - \alpha_v = 1$ then
        return $\alpha_v$
    end if
    if $\tilde{B}_\ell[i] = 0$ then
        $i \leftarrow \mathtt{rank}_0(\tilde{B}_\ell, i)$
    else
        $i \leftarrow \mathtt{rank}_1(\tilde{B}_\ell, i)$
    end if
    return acc($\ell$+1, $i$)
```

```
rnk($\ell, a, i$)
    if $\omega_v - \alpha_v = 1$ then
        return $i - C[a]$
    end if
    if $a < 2^{\lceil \lg(\omega_v - \alpha_v) \rceil - 1}$ then
        $i \leftarrow \mathtt{rank}_0(\tilde{B}_\ell, i)$
    else
        $i \leftarrow z_\ell + \mathtt{rank}_1(\tilde{B}_\ell, i)$
    end if
    return rnk($\ell$+1, $a, i$)
```

```
sel($a, j$)
    $\ell \leftarrow \lceil \lg \sigma \rceil$, $d \leftarrow \lceil \lg \sigma \rceil - \ell + 1$
    $j \leftarrow C[a] + j$
    while $\ell \geq 0$ do
        if $a \bmod 2^d = 0$ then
            $j \leftarrow \mathtt{select}_0(\tilde{B}_\ell, j)$
        else
            $j \leftarrow \mathtt{select}_1(\tilde{B}_\ell, j - z_\ell)$
        end if
        $\ell \leftarrow \ell - 1$, $d \leftarrow d + 1$
    end while
    return $j$
```

$$\begin{array}{lc}
\ell = 1 & \ldots, c_8, c_{12}, c_{32}, c_{48} \ldots \\
\ell = 2 & \ldots, c_8, c_{12}, \ldots | \ldots, c_{32}, c_{48} \ldots \\
\ell = 3 & \ldots, c_{32} \ldots | \ldots, c_8, c_{12}, \ldots, c_{48} \ldots \\
\ell = 4 & \ldots, c_{32}, \ldots, c_8, \ldots, c_{48} | \ldots, c_{12}, \ldots \\
\ell = 5 & \ldots, c_{16}, \ldots, c_8, \ldots, c_{48}, \ldots, c_{12} \ldots | \ldots \\
\ell = 6 & \ldots, c_{32}, c_{48}, \ldots | \ldots
\end{array}$$

Figure 5: Example of a sequence of canonical codes along wavelet matrix levels, showing that the leaves do not span a contiguous area. The the vertical bar "|" marks the points z_ℓ.

it reduces the redundancy from $o(n \lg \sigma)$ to $o(n(H_0(S)+1))$ (albeit a small $O(\sigma \lg n)$-bit space term is added to hold the Huffman model); (4) the bitmap compression can be additionally combined with the Huffman shape, obtaining further compression (yet higher time).

The idea is the same as in Section 3: Arrange the codes so that all the leaves are grouped to the right of the bitmaps $\tilde{B}_\ell$. However, because of the reordering of nodes produced by the wavelet matrix, the use of canonical Huffman codes does not guarantee that the leaves of the same level are contiguous. In the wavelet matrix, the position of a code c in $\tilde{B}_{\ell+1}$ depends only on the position of c in $\tilde{B}_\ell$ and on the bit of c in that level, $c[\ell]$. Figure 5 illustrates an example of a canonical set of codes where the first 16 shortest codewords take values from 00000 to 01111 and the remaining 32 from 100000 to 111110. The figure shows the relative positions of the codes at successive levels of the wavelet matrix for a sequence $\ldots c_8, c_{12}, c_{32}, c_{48} \ldots$, where $c_8 = 01000$, $c_{12} = 01100$, $c_{32} = 100000$, and $c_{48} = 110000$. As we can see, codes c_8 and c_{12} finish at level 5 but they are not contiguous since there is a c_{48} between them.

Algorithm 7 Range search algorithms on the wavelet matrix: **count**$(0, x_1, x_2, y_1, y_2)$ returns `count`(P, x_1, x_2, y_1, y_2) on the wavelet tree of sequence P; and **report**$(0, x_1, x_2, y_1, y_2)$ outputs all those y, where a point with coordinate $y_1 \leq y \leq y_2$ appears in $P[x_1, x_2]$. For simplicity we have omitted the computation of $[\alpha_v, \omega_v)$.

```
count($\ell, x_1, x_2, y_1, y_2$)
  if $x_1 > x_2 \vee [\alpha_v, \omega_v] \cap [y_1, y_2] = \emptyset$ then
    return 0
  else if $[\alpha_v, \omega_v] \subseteq [y_1, y_2]$ then
    return $x_2 - x_1 + 1$
  else
    $x_1^l \leftarrow \mathtt{rank}_0(\tilde{B}_\ell, x_1 - 1) + 1$
    $x_2^l \leftarrow \mathtt{rank}_0(\tilde{B}_\ell, x_2)$
    $x_1^r \leftarrow x_1 - x_1^l + 1$, $x_2^r \leftarrow x_2 - x_2^l$
    return count($\ell+1, x_1^l, x_2^l, y_1, y_2$)
          +count($\ell+1, x_1^r, x_2^r, y_1, y_2$)
  end if
```

```
report($v, x_1, x_2, y_1, y_2$)
  if $x_1 > x_2 \vee [\alpha_v, \omega_v] \cap [y_1, y_2] = \emptyset$ then
    return
  else if $\omega_v - \alpha_v = 1$ then
    output $\alpha_v$
  else
    $x_1^l \leftarrow \mathtt{rank}_0(\tilde{B}_\ell, x_1 - 1) + 1$
    $x_2^l \leftarrow \mathtt{rank}_0(\tilde{B}_\ell, x_2)$
    $x_1^r \leftarrow x_1 - x_1^l + 1$, $x_2^r \leftarrow x_2 - x_2^l$
    report($\ell+1, x_1^l, x_2^l, y_1, y_2$)
    report($\ell+1, x_1^r, x_2^r, y_1, y_2$)
  end if
```

We require a different mechanism to design an optimal prefix-free code that guarantees that, under the shuffling rules of the wavelet matrix, all the leaves at any level form a contiguous area to the right of the bitmap.

We start by studying how the wavelet matrix sorts the codes at each level. Consider a pair of codes $c_1[1, \ell_1]$ and $c_2[1, \ell_2]$. Depending on their bits at a given level ℓ of the wavelet matrix, two cases are possible: (a) $c_1[\ell] = c_2[\ell]$ and then the relative positions of c_1 and c_2 stay the same at level $\ell + 1$, or (b) $c_1[\ell] \neq c_2[\ell]$ and then their relative positions in level $\ell + 1$ depend on the relation between $c_1[\ell]$ and $c_2[\ell]$. This yields the following proposition:

Proposition 2. *In a wavelet matrix, given any pair of codes c_1 and c_2, c_1 appears before(after) c_2 in $\tilde{B}_\ell$ if, for some $0 \leq i < \ell$, it holds $c_1[\ell-i, \ell-1] = c_2[\ell-i, \ell-1]$ and $c_1[\ell-i-1] = 0(1) \neq c_2[\ell-i-1]$.*

Proof. If $c_1[\ell - i, \ell - 1] = c_2[\ell - i, \ell - 1]$, then c_1 and c_2 transitively keep their relative positions from level $\ell - i$ to level ℓ. Instead, $c_1[\ell - i - 1] \neq c_2[\ell - i - 1]$ makes their ordering in level $\ell - i$ dependent only on how $c_1[\ell - i - 1]$ and $c_2[\ell - i - 1]$ compare to each other. □

As a second step, assume we want to design a set of fixed-length codes $\{c_a,\ a \in [0, \sigma)\}$ such that $c_a < c_b$ iff the area of c_a is before that of c_b in $\tilde{B}_{\lceil \lg \sigma \rceil}$. That is, we want the codes to be listed in order in the last level. Let $inv : \{0,1\}^{\mathbb{N}^+} \times \mathbb{N}^+ \to \{0,1\}^{\mathbb{N}^+}$ be defined as $inv(c[1, \ell], \ell) = c^{-1}[1, \ell]$, where $c^{-1}[i] = c[\ell - i + 1]$ for all $1 \leq i \leq \ell$. That is, $inv(c, \ell)$ takes number c as a codeword of ℓ bits and returns the code obtained by reading c backwards. Then, the following proposition holds:

Proposition 3. *Given any two values $i, j \in [0, \sigma)$ where $i < j$, code $inv(i, \lceil \lg \sigma \rceil)$ is located to the left of code $inv(j, \lceil \lg \sigma \rceil)$ in the bitmap $\tilde{B}_{\lceil \lg \sigma \rceil}$ of a wavelet matrix that uses such codes.*

Proof. Let $\tau_i = inv(i, \lceil \lg \sigma \rceil)$ and $\tau_j = inv(j, \lceil \lg \sigma \rceil)$. If τ_i and τ_j do not share any common suffix, then their relative positions in $\tilde{B}_{\lceil \lg \sigma \rceil}$ depend only on their last bit and the relation is given by that bit. Otherwise, τ_i and τ_j share a common suffix of length $\lceil \lg \sigma \rceil - \delta + 1 \in [1, \lceil \lg \sigma \rceil]$, that is, $\tau_i[\delta, \lceil \lg \sigma \rceil] = \tau_j[\delta, \lceil \lg \sigma \rceil]$. Then, according to Proposition 2, τ_i is before τ_j iff $\tau_i[\delta] < \tau_j[\delta]$. In both cases the relation is given by the last distinct bit of the codes, or the first if they are read backwards. Since the codes are of the same length, comparing by the first distinct bit is equivalent to comparing numerically. That is, τ_i is before τ_j iff $inv(\tau_i, \lceil \lg \sigma \rceil) < inv(\tau_j, \lceil \lg \sigma \rceil)$. In turn, since $inv(inv(c,\ell),\ell) = c$, this is equivalent to $i < j$. $\square$

The proposition gives a way to force a desired order in a set of fixed-length codes: Given symbols $a \in [0, \sigma)$, we can assign them codes $c_a = inv(a, \lceil \lg \sigma \rceil)$ to ensure that the areas become ordered in $\tilde{B}_{\lceil \lg \sigma \rceil}$. As a side note, we observe that we could have retained the symbol order natively in the wavelet matrix if we had chosen to decompose the symbols from their least to their most significant bit, and not the other way (in this case the wavelet matrix is actually radix-sorting the values). This brings problems in the extended variants, however, because the resulting range of codes has unused entries if σ is not a power of 2. For example, consider alphabet $0, 1, 2, 3, 4 = 000, \ldots, 100$; after reversing the bits we obtain numbers $0, 1, 2, 4, 6$, so we need to allocate 7 cells for C instead of 5. The size of C can double in the worst case. We cannot either directly use the idea of reversing the canonical Huffman codes, because the codes could not be prefix-free anymore. A more sophisticated scheme, based on Proposition 3, is required.

Assume we have obtained the desired code lengths ℓ_a, as well as the array $nCodes$ from the canonical Huffman construction. We generate the final Huffman tree in levelwise order. The simplest description is as follows. We start with a set of valid codes $\mathcal{C} = \{0, 1\}$ and level $\ell = 1$. At each level ℓ, we remove from $\mathcal{C}$ the $nCodes[\ell]$ codes c with minimum $inv(c, \ell)$ value. The removed nodes are assigned to the $nCodes[\ell]$ symbols that require codes of length ℓ. Now we replace each code c remaining in $\mathcal{C}$, by two new codes, $c : 0$ and $c : 1$, and continue with level $\ell + 1$. It is clear that this procedure generates a prefix-free set of codes that, when reversed, satisfy that the codes finishing at a level are smaller than those that continue.

It is not hard to see that the total cost of this algorithm is linear. There are two kind of codes inserted in $\mathcal{C}$: those that will be chosen for a code and those that will not. There are exactly σ nodes of the first class, whereas for each node of the second class we insert other two codes in $\mathcal{C}$. Therefore the total number of codes ever inserted in $\mathcal{C}$ adds up to $O(\sigma)$. The codes to use at each level ℓ can be obtained by linear-time selection over the set of codes just extended (sorting codes by $inv(c, \ell)$), thus adding up to $O(\sigma)$ time as well.

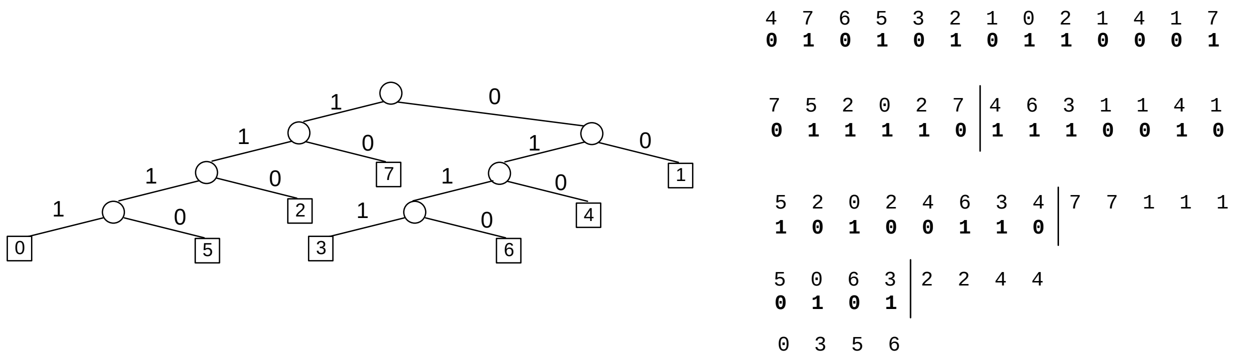
Figure 6 gives an example of the construction.

6 Experimental Results

Our implementations build over the wavelet tree implementations of LIBCDS, a library implementing several space-efficient data structures.[3] For each wavelet tree/matrix variant we present two versions, `CM` and `RRR`. The first one corresponds to using the the `rank`/`select` enabled bitmap implementation [34] of the proposals of Clark [19] and Munro [49], choosing 5% space overhead over the plain bitmap. The second version, `RRR`, corresponds to using the bitmap implementation [20] of the compressed bitmaps of Raman, Raman and Rao [61]. The variants compared are the following:

- `WT`: standard pointer-based wavelet tree;

[3] `https://github.com/fclaude/libcds`



```
4  7  6  5  3  2  1  0  2  1  4  1  7
0  1  0  1  0  1  0  1  1  0  0  0  1

7  5  2  0  2  7 | 4  6  3  1  1  4  1
0  1  1  1  1  0 | 1  1  1  0  0  1  0

5  2  0  2  4  6  3  4 | 7  7  1  1  1
1  0  1  0  0  1  1  0 |

5  0  6  3 | 2  2  4  4
0  1  0  1 |

0  3  5  6
```

Figure 6: On the left, the Huffman tree resulting from our code reassignment algorithm on the running example. On the right, the resulting Huffman-shaped wavelet matrix.

- `WTNP`: the (extended) pointerless wavelet tree ("No Pointers");
- `WM`: the (extended) wavelet matrix (Section 4);
- `HWT`: the Huffman-shaped standard pointer-based wavelet tree;
- `HWTNP`: the Huffman-shaped extended levelwise wavelet tree (Section 3);
- `HWM`: the Huffman-shaped (extended) wavelet matrix (Section 5);
- `AP`: the alphabet-partitioned data structure of Barbay et al. [4], which is the best state-of-the-art alternative to wavelet trees.

These names are composed with the bitmap implementations by appending the bitmap representation name. For example, we call `WT-RRR` the standard pointer-based wavelet tree with all bitmaps represented with Raman, Raman and Rao's compressed bitmaps. `AP` uses always `CM` bitmaps, which is the best choice for this structure.

Note that all the pointerless structures use the array C. The extended versions generally achieve space very close to the strict ones and perform much faster.

6.1 Datasets

In order to evaluate the performance of `access`, `rank` and `select`, we use four different datasets:[4]

- `ESWiki`: Sequence of word identifiers generated by stemming the Spanish Wikipedia[5] with the Snowball algorithm. The sequence has length n = 200,000,000, alphabet size σ = 1,634,145, and zero-order entropy H_0 = 11.12. This sequence can be used to simulate a positional inverted index [20, 2, 14].

[4]Left at `http://lbd.udc.es/research/ECWTLA`

[5]`http://es.wikipedia.org` dated 03/02/2010.

- `BWT`: The Burrows-Wheeler transform (BWT) [16] of `ESWiki`. The length and size of the alphabet, as well as the zero-order entropy, match those of `ESWiki`. However, `BWT` has a much lower high-order entropy [48]. Many full-text compressed self-indexes [28, 29, 52] use the BWT of the text they represent.

- `Indochina`: The concatenation of all adjacency lists of Web graph `Indochina2004`, available at the WebGraph project.[6] The length of the sequence is $n = 100{,}000{,}000$, the alphabet size $\sigma = 2{,}705{,}024$, and the entropy is $H_0 = 15.69$. This representation has been used to support forward and backward traversals on the graph [20, 21].

- `INV`: Concatenation of inverted lists for a random sample of 2,961,510 documents from the English Wikipedia.[7] This sequence has length $n = 338{,}027{,}430$ and its alphabet size is $\sigma = 2{,}961{,}510$. From this sequence we extract the first $n = 180{,}000{,}000$ elements with an alphabet of size $\sigma = 1{,}590{,}398$ and an entropy of $H_0 = 19.01$. This sequence has been used to simulate document inverted indexes [58, 31].

In order to evaluate the range search performance over discrete grids, we use the following three datasets formed by synthetic and real collections of MBRs (Minimum Bounding Rectangles of objects). We insert the two opposite corners of each MBR as points in our dataset.

- `Zipf`: A synthetic collection of 1,000,000 MBRs with a Zipfian distribution (world size = $1{,}000 \times 1{,}000$, $\rho = 1$).[8]

- `Gauss`: A synthetic collection of contains 1,000,000 MBRs with a Gaussian distribution (world size = $1{,}000 \times 1{,}000$, $\mu = 500, \sigma = 200$).[8]

- `Tiger`: A real collection of 2,249,727 MBRs from California roads, available at the U.S. Census Bureau.[9]

For range searches we cannot use Huffman compression, because the order of the symbols is not maintained at the leaves. `AP` also shuffles the alphabet, so it cannot be used in this scenario. Extended variants are not a good option either, because in this case it holds $\sigma = n$. Thus we test only the strict variants of `WTNP` and `WM`.

6.2 Measurements

To measure performance we generated 100,000 inputs for each query and averaged their execution time, running each query 10 times. The $\texttt{access}(S, i)$ queries were generated by choosing positions i uniformly at random in $[1, n]$. Queries $\texttt{rank}_a(S, i)$ were generated by choosing i uniformly at random, and then setting $a = S[i]$. Each $\texttt{select}_a(S, j)$ query was generated by first choosing a position i at random in $[1, n]$, then setting $a = S[i]$, and finally choosing j at random in $[1, \texttt{rank}_a(S, n)]$.

[6] `http://law.dsi.unimi.it`

[7] `http://en.wikipedia.org`

[8] `http://lbd.udc.es/research/serangequerying`

[9] `http://www.census.gov/geo/www/tiger`

The resulting distribution is the most common in applications, and it obtains the $O(H_0(S)+1)$ average time performance in the Huffman-shaped variants.

To measure the performance on point grids, for synthetic collections we generate sets of queries covering 0.001%, 0.01%, 0.1%, and 1% of the grid area. The sets contain 1,000 queries, each with a ratio between both axes varying uniformly at random between 0.25 and 2.25. For the real data set `Tiger`, we use as queries the following four collections (also available for downloading at the Web site of `Tiger`): `Block` (groups of buildings), `BG` (block groups), `SD` (elementary, secondary, and unified school districts), and `COUSUB` (country subdivisions).

The machine used is an Intel(R) Xeon(R) E5620 running at 2.40GHz with 96GB of RAM memory. The operating system is GNU/Linux, Ubuntu 10.04, with kernel 2.6.32-33-server.x86_64. All our implementations use a single thread and are coded in `C++`. The compiler is `gcc` version 4.4.3, with `-O9` optimization.

6.3 Results on Sequences

Figures 7 to 9 show the time and space for the different data structures and configurations for `access`, `rank` and `select` queries. The black vertical bar on the plots shows the value of H_0. The bitmaps are parametrized by setting their sampling values to 32, 64, and 128. In the case of `AP`, these bitmap samplings are combined with permutation samplings 4, 16, and 64, respectively, and all are run with $\ell_{min} = 10$, as in previous work [4].

Space. We start by discussing the space usage, which we measure in bits per symbol (bps). First we note that the `WM` variants use always the same space as the corresponding `WTNP` variants (while being faster, as we discuss soon). The space of `WTNP-CM` and `WM-CM` is obviously close to $\lceil \lg \sigma \rceil$ bps. The extra space incurred by `WT-CM` is the overhead of the wavelet tree pointers, and is roughly proportional to σ/n (times some implementation-dependent constant). This amounts to nearly 4 bps in `ESWiki` and `BWT`, but 3.5 times more (14 bps) in `Indochina`, as expected from its larger alphabet size, and again 4 bps in `INV`. On the other hand, the space of `HWTNP-CM` and `HWM-CM` is always close to H_0 bits per symbol, plus a small extra to store the Huffman model. The space overhead of `HWT-CM` on top of those corresponds, again, to the wavelet tree pointers.

The sampling parameter affects more sharply the `RRR` variants, as they store more data per sample. The difference between `WTNP-RRR` or `WM-RRR` and `WT-RRR` is also proportional to σ/n, but this time the constant is higher because the `RRR` implementation needs more constants to be stored per bitmap (i.e., per wavelet tree node). Thus the penalty is 6 bps on `ESWiki` and `BWT`, 21 bps (3.5 times more) on `Indochina`, and 7 bps on `INV`. The same differences can be observed between `HWTNP-RRR` or `HWM-RRR` and `HWT-RRR`. We return later to the fact that `HWM-RRR` takes more space than `HWTNP-RRR` on `Indo` and `INV`.

Finally, how `WTNP-RRR`/`WM-RRR` and `HWTNP-RRR`/`HWM-RRR` compare to `HWTNP-CM`/`WM-CM` depends strongly on the type of sequence. In general, `RRR` compression achieves the zero-order entropy as an upper bound, but it can reach much less when the sequence has local regularities. On the other hand, `RRR` representation poses an additive overhead of 27% of $\lg \sigma$, which corresponds to the $o(n \lg \sigma)$ overhead in this implementation [20]. When combining Huffman and bitmap compression, this 27% overhead acts over H_0 and not over $\lg \sigma$, which brings it down, but on the other hand we must add the overhead of storing the Huffman model. On `ESWiki`, which has no special properties, the 27% overhead is around 5.7 bps, showing that RRR compression reaches around 8.3 bps, well below H_0. When combining with Huffman compression, this overhead becomes 14%, that is, nearly 3 bps. Added to the 8.3 bps and to the 1 bps of the Huffman model overhead, we still get slightly more space than plain Huffman compression, which is the best choice and reaches only 10% overhead over the zero-order entropy.

The picture changes when we consider BWT. The Burrows-Wheeler transform of ESWiki boosts its higher-order compressibility [48], which is captured by RRR compression [44], making RRR compression reach the same space of Huffman compression, despite its 27% space overhead. When combining both compressions, the result breaks the zero-order entropy barrier by more than 10% and becomes the best choice in terms of space.

RRR gives another surprising result on Indochina and INV, where bitmap compression alone is more space-effective than in combination with Huffman compression, and breaks the zero-order entropy by a large margin. This cannot be explained by high-order compressibility, as in this case the combination with Huffman would not harm. This behavior corresponds to the special nature of these sequences: the adjacency lists of the graph and the inverted lists are sorted in increasing order. Long increasing sequences induce long runs of 0s and 1s in the bitmaps of the wavelet trees and matrices. Those are retained in deeper levels when our partitioning by the most significant bit is used.[10] The Huffman algorithm, instead, combines the nodes in unpredictable ways and destroys those long runs. Still, our Huffman algorithm maintains the order between those symbols whose codewords have the same length, and thus the impact of this reordering is not as high as it could be. Instead, the Huffman wavelet matrix completely reshuffles the symbols. As a result, for example, the space of HWM-RRR exceeds that of HWTNP-RRR by around 5 bps on Indo and 6–7 bps on INV.

Time. The time results are rather consistent across collections. Let us first consider operation access. If we start considering the variants that do not use Huffman compression, we have that WT-RRR is about 10%–25% slower than WT-CM, which is explained by a more complex implementation [20]. Instead, the pointerless variant, WTNP-CM, is 20%–25% slower (recall that, in their extended variant, these require twice the number of rank operations, but locality of reference makes them faster than twice the cost of one rank operation). However, WTNP-RRR is about 40% slower than WT-RRR, as the rank operation is slower and its higher number impacts more on the total time (but still locality of reference makes the percentage much less than 100%). The wavelet matrix, instead, carries out the same number of rank operations than the pointer-based wavelet tree, so this time penalty disappears. Actually, WM-CM is 8%–14% *faster* than WT-CM, and WM-RRR is up to 4% faster than WT-RRR. This may be due to less memory usage, which increases locality of reference. Finally, the use of Huffman compression improves times by about $H_0/\lg\sigma$, as expected: times are reduced to about 50%–60% on ESWiki and BWT, to about 65%–85% on Indochina, and there is almost no reduction on INV.

The situation is basically the same for operation rank, as expected from the algorithms. The times are usually slightly lower because it is not necessary to access the bitmaps as we descend. The use of the wavelet matrix still gives essentially the same time (and even slightly faster) than a pointer-based wavelet tree, and the use of Huffman-shaped trees reduces the times by the same factors as for access, as expected.

[10]This is another advantage over using the least significant bit, which would break the runs faster.

The times of operation select show less difference between the standard and the pointerless variants, because performing one extra rank operation is less relevant compared to the original (slower) select operation on the bitmaps. One can see that WTNP-CM is 30%–40% slower than WT-CM and that WTNP-RRR is 35%–50% slower than WT-RRR. The difference between plain and compressed bitmaps does not vary much, on the other hand: WT-RRR is 25%–30% slower than WT-CM. What is more surprising is that the wavelet matrix is clearly slower than the pointer-based wavelet trees: WM-CM is 10%–15% slower than WT-CM and WM-RRR is 20%–30% slower than WT-RRR. The reason is that the implementations of select [34, 20] proceed by binary search on the sampled values, thus their cost has in practice a component that is logarithmic on the bitmap length. The bitmaps on the wavelet tree nodes are shorter than n, whereas in the wavelet matrix (and the pointerless wavelet tree) they are always of length n. Indeed, the wavelet matrix is faster than the pointerless wavelet tree: WM-CM is 20%–25% faster than WTNP-CM and WM-RRR is 12%–15% faster than WTNP-RRR. Once again, the use of Huffman reduces all the times by about the same space fraction obtained by zero-order compression.

Bottom line. On ESWiki, where zero-order compression is the dominant space factor, our Huffman-shaped wavelet matrix, HWM-CM, obtains the best space (only 10% off the zero-order entropy) and the best time, by a good margin.

On BWT, where higher-order compression is exploited by RRR, the space-time tradeoff map is dominated by the combination of HWM-RRR (minimum space) and HWM-CM (minimum time), the two variants of our Huffman-shaped wavelet matrix. The former breaks the zero-order entropy barrier by about 10%.

On Indochina and INV, where RRR achieves space gains that are only degraded by Huffman compression, the dominant techniques are variants of the wavelet matrix: WM-RRR (least space) and HWM-CM (least time). The former takes about 75% of the zero-order entropy.

Summarizing, the wavelet matrix variants obtain the same space of the pointerless wavelet trees, but they operate in about 65% of their time, reaching basically the same performance of the pointer-based variants but much less space. As a result, they are always the dominant technique. Which variant is the best, HWM-CM, HWM-RRR or WM-RRR, depends on the nature of the collection.

The comparison with AP is interesting. In collections similar to ESWiki, Barbay et al. [4] show that AP generally achieves the best space and time among the alternatives WTNP-RRR, WTNP-CM, WT-CM, and WT-RRR, thus becoming an excellent choice in that group. The pointerless Huffman-shaped alternatives, however, clearly outperform AP in space: pointerless Huffman compression, and in particular Huffman wavelet matrices, improve upon the old wavelet tree alternatives in both space and time, using much less space than AP. Still, AP is a faster representation, only slightly faster in operations access and rank, and definitely faster in operation select. The other collections also demonstrate that wavelet trees and matrices can exploit other compressibility features of the sequences apart from H_0, whereas AP is blind to those (this is also apparent in their experiments [4], even using the basic wavelet tree variants).

Note that, for simplicty and uniformity, we have built our prototypes and experiments on the implementations of Libcds. Other ones could be tried. For example, there exists an alternative RRR implementation [57] that, at the price of some (further) increase in time, reduces the 27% space overhead to 10% on a 64-bit architecture. This would multiply the space of all the RRR alternatives by up to 0.86, making them even more space-attractive (and making `HWM-RRR` the most space-efficient choice on `ESWiki`). On the other collections, our conclusions above would not change. Moreover, on those collections the space fraction is likely to be higher than 0.86, as these benefit from locality in compression. The alternative implementation [57] captures zero-order entropy of 63-bit chunks, and thus it is less local than the implementation we used, which takes 15-bit chunks. As another example, one could add more efficient implementations of bitwise `select` [60, 67, 57], which would make the impact of the better wavelet matrix organization more clear.

6.4 Results on Point Grids

Figures 10 and 11 show the performance of `WTNP` and `WM` for `count` and `report` queries, respectively. It turns out that, in the first level of each wavelet tree, the number of zeros and ones is highly unbalanced when the grid size is far from the next power of 2. This makes the entropy of the first bitmap rather low, whereas the other bitmaps are more balanced. On the other hand, the range search algorithms spend just a few `rank` operations on the first bitmap. To take advantage of this feature, we compress the bitmap of the first level of both data structures, `WTNP` and `WM`, with `RRR` and with a sampling of 32. The rest of bitmaps are represented using `CM` with sampling rates of 32, 64, and 128.

In both figures 10 and 11 we append to the name of the data structure the name of the query set. This takes values in $\{Q0001, Q001, Q01, Q1\}$ in case of synthetic collections. In case of the real collection `Tiger`, it takes values in {BLock, BG, SD, COUSUB}.

The results for the counting queries shows that the time worsens as the query are less selective. The wavelet matrix is always faster than the pointerless wavelet tree, while using the same space. The difference in time is proportional to the cost for each selectivity, but additive with respect to the sampling. For example, it becomes about 25% faster when using the most space. We note in passing that the space is basically 21 bps for the synthetic spaces and 23 bps for the `Tiger` dataset, which is essentially $\lg \sigma = \lg n$.

In the case of reporting queries, we show the time per reported item, which decreases as the query is less selective. Once again the wavelet matrix is faster than the pointerless wavelet tree, albeit this time by a smaller margin: usually below 10%.

7 Conclusions

The levelwise wavelet tree [43, 45], designed to avoid the $O(\sigma \lg n)$ space overhead of standard wavelet trees [35], was unnecessarily slow in practice. We have redesigned this data structure so that its time overhead over standard wavelet trees is significantly lower. The result, dubbed *wavelet matrix*, enjoys all the good properties of levelwise wavelet trees but performs significantly faster in practice. It requires $n \lg \sigma + o(n \lg \sigma)$ bits of space, and can be built in $O(n \lg \sigma)$ time and almost in-place. We have also adapted pointerless Huffman-shaped wavelet trees to become Huffman-shaped wavelet matrices. This required a nontrivial redesign of the variable-length code assignment mechanism. Our experimental results show that the compressed wavelet matrix dominates the space/time tradeoff map for all the real-life sequences we considered, also outperforming in most cases other structures designed for large alphabets [4]. We also showed that the wavelet matrix is the best choice to represent point grids that support orthogonal range queries.

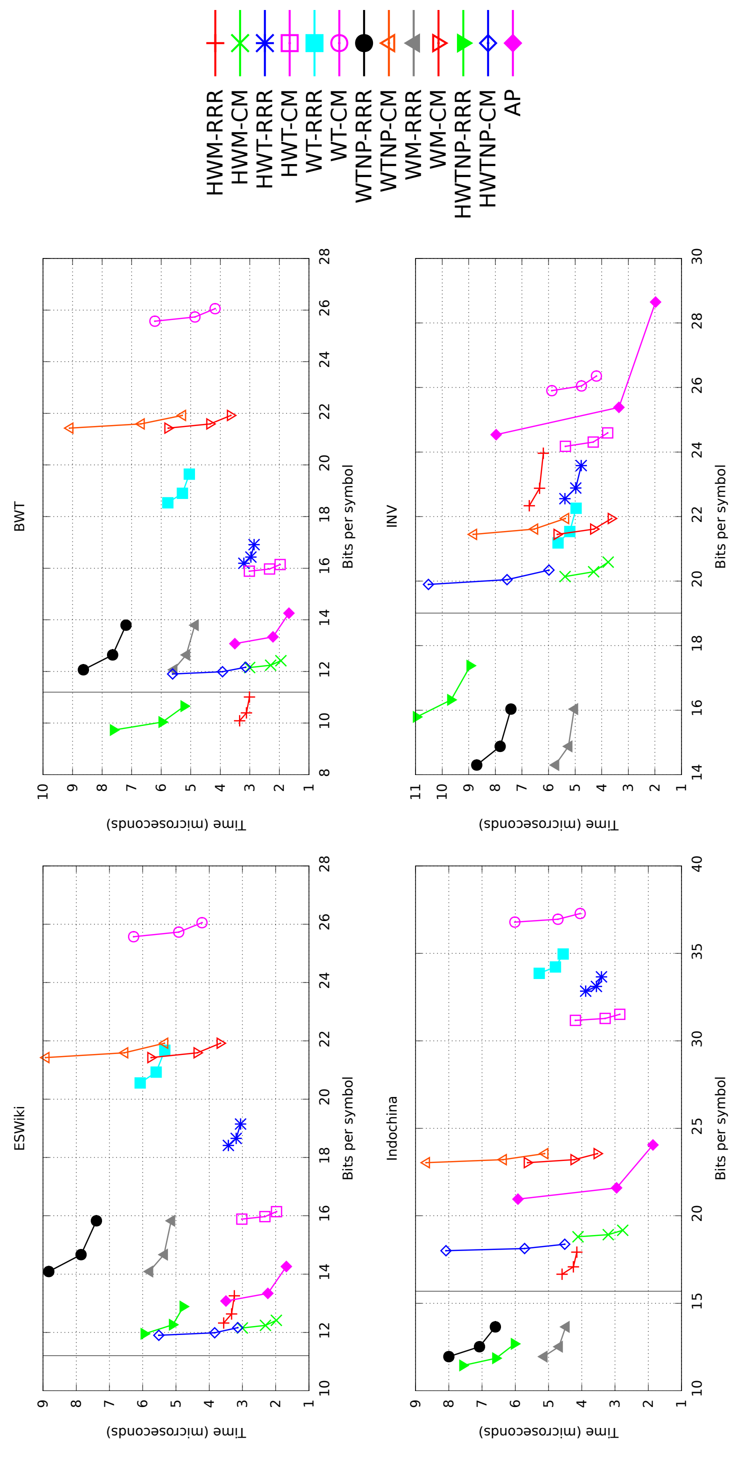


Figure 7: Running time per `access` query over the four datasets.

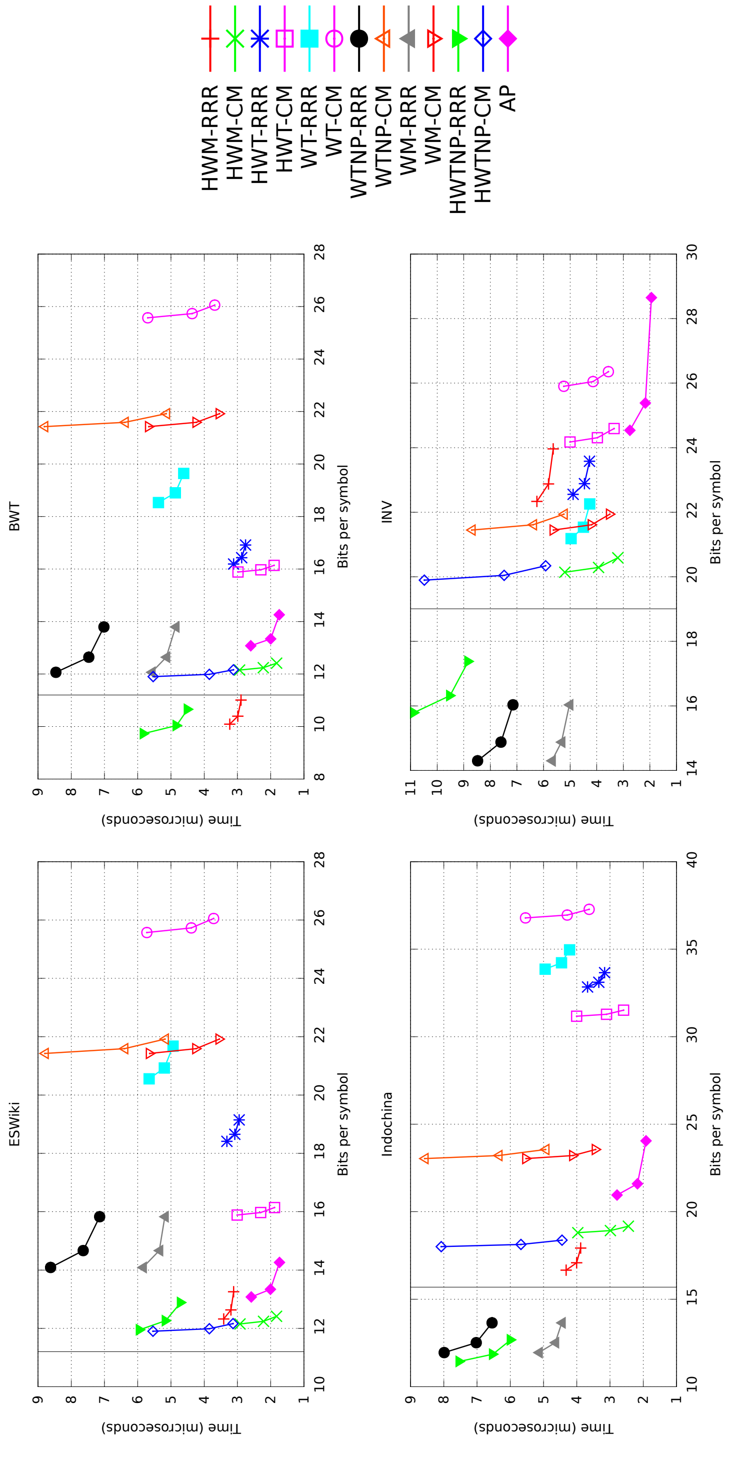


Figure 8: Running time per `rank` query over the four datasets.

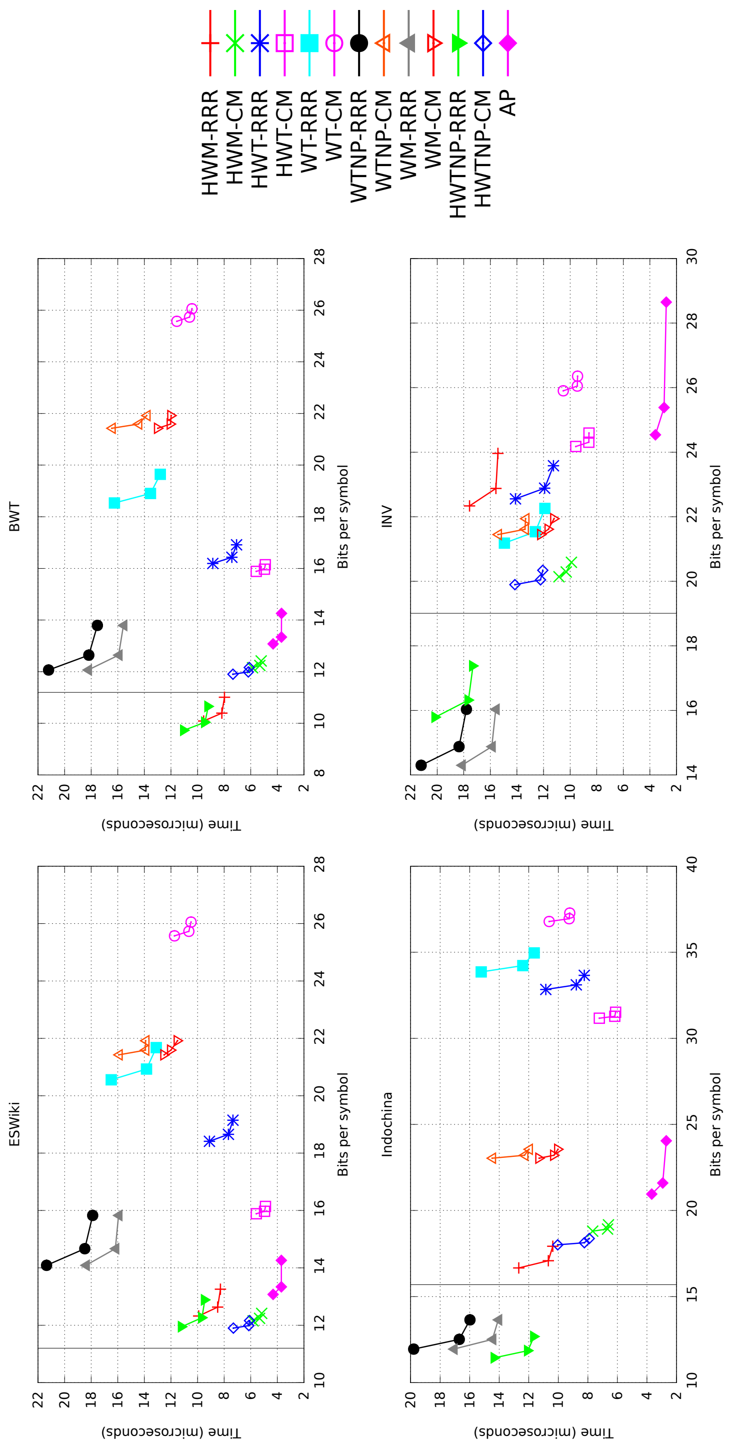


Figure 9: Running time per `select` query over the four datasets.

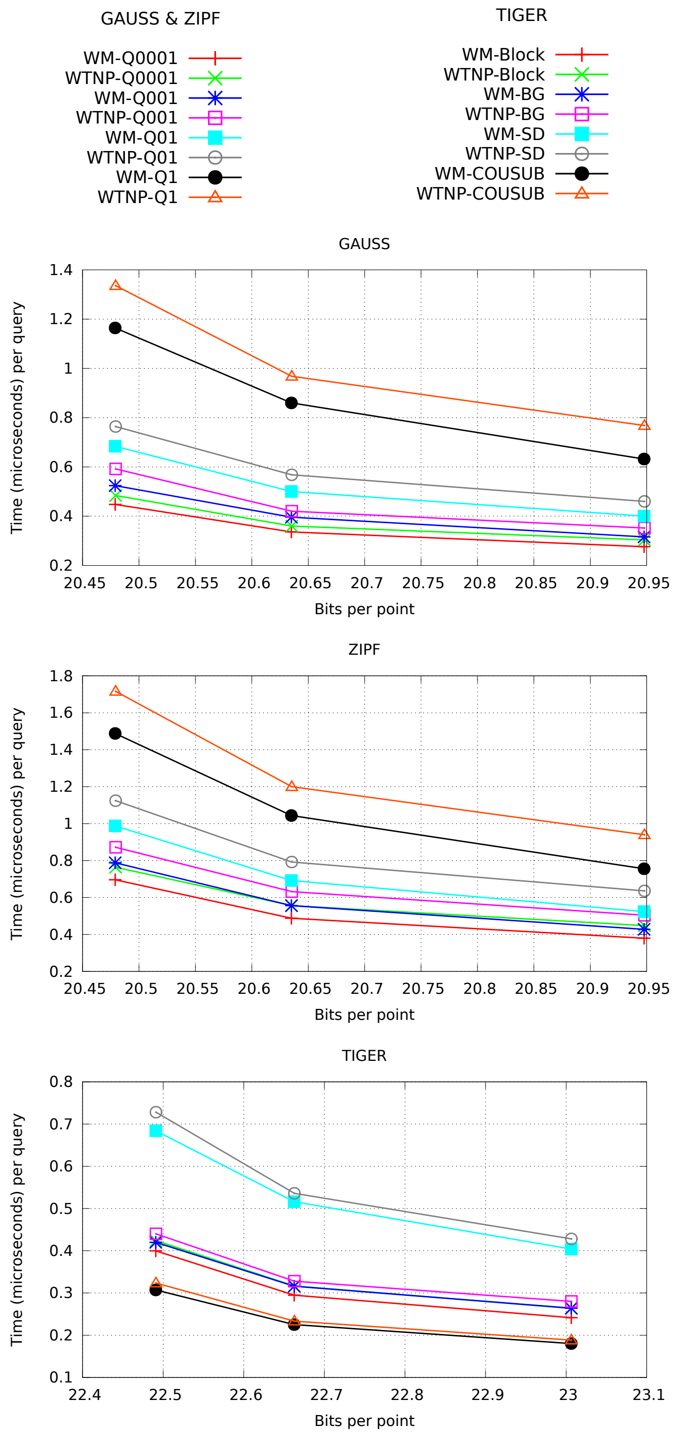


Figure 10: Running time per `count` query over the three datasets.

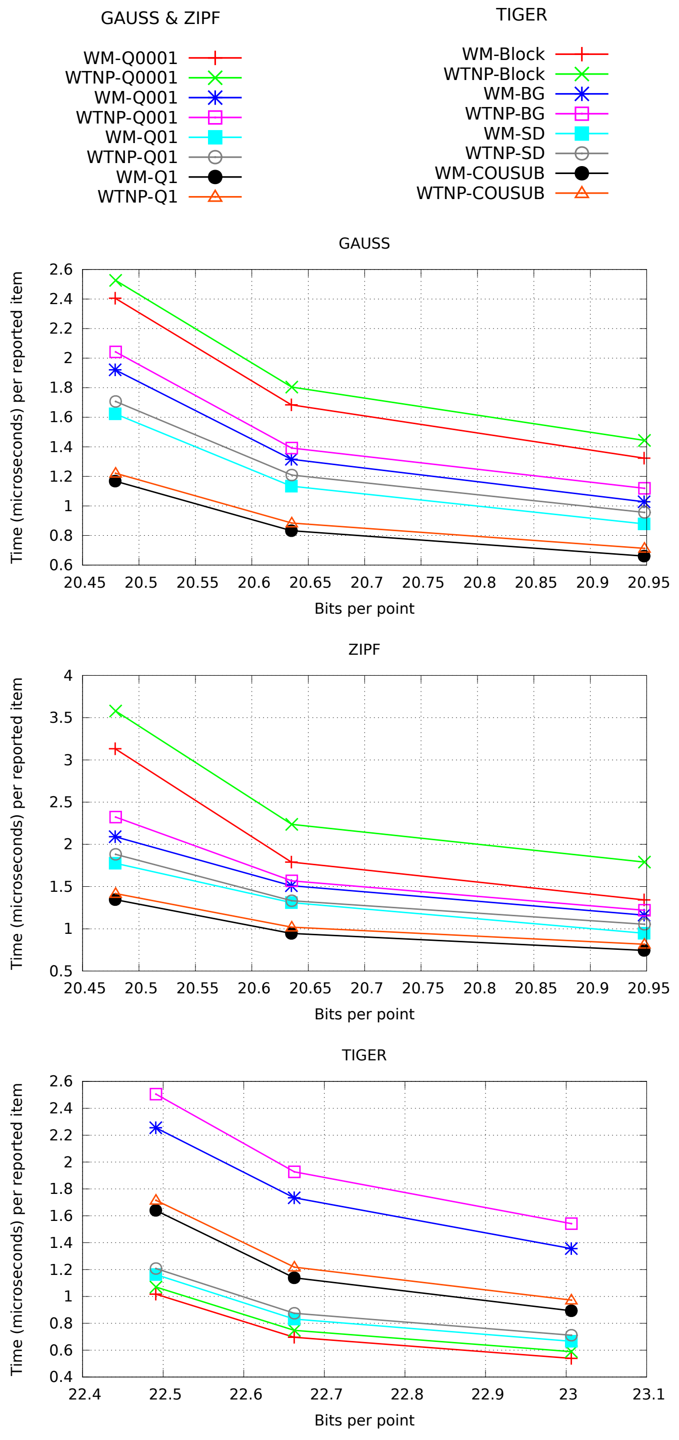


Figure 11: Running time of `report` query over the three datasets.

An interesting future work is to adapt multiary wavelet trees [29] to wavelet matrices. The only difference is that, instead of a single accumulator z_ℓ per level, we have an array of $\rho-1$ accumulators in a ρ-ary wavelet matrix. As the useful values for ρ are $O(\lg n)$, the overall space is still negligible, $O(\lg^2 n \lg \sigma)$. The real challenge is to transalte the reduction in depth into a reduction of actual execution times.

Dynamic wavelet trees [46, 38, 59] can immediately be translated into wavelet matrices. It would be interesting to consider newer, theoretically more efficient dynamic versions [54], and obtain practically efficient implementations over wavelet matrices.

Acknowledgement. Thanks to Daisuke Okanohara for useful comments.

References

[1] D. Arroyuelo, F. Claude, S. Maneth, V. Mäkinen, G. Navarro, K. Nguyễn, J. Sirén, and N. Välimäki. Fast in-memory XPath search over compressed text and tree indexes. In *Proc. 26th IEEE International Conference on Data Engineering (ICDE)*, pages 417–428, 2010.

[2] D. Arroyuelo, S. González, and M. Oyarzún. Compressed self-indices supporting conjunctive queries on document collections. In *Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 43–54, 2010.

[3] J. Barbay, L.C. Aleardi, M. He, and J.I. Munro. Succinct representation of labeled graphs. *Algorithmica*, 62(1-2):224–257, 2012.

[4] J. Barbay, F. Claude, T. Gagie, G. Navarro, and Y. Nekrich. Efficient fully-compressed sequence representations. *Algorithmica*, 69(1):232–268, 2014.

[5] J. Barbay, F. Claude, and G. Navarro. Compact rich-functional binary relation representations. In *Proc. 9th Latin American Symposium on Theoretical Informatics (LATIN)*, LNCS 6034, pages 170–183, 2010.

[6] J. Barbay, T. Gagie, G. Navarro, and Y. Nekrich. Alphabet partitioning for compressed rank/select and applications. In *Proc. 21st Annual International Symposium on Algorithms and Computation (ISAAC)*, LNCS 6507, pages 315–326 (part II), 2010.

[7] J. Barbay, A. Golynski, I. Munro, and S. Srinivasa Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. *Theoretical Computer Science*, 387(3):284–297, 2007.

[8] J. Barbay, M. He, I. Munro, and S. Srinivasa Rao. Succinct indexes for strings, binary relations and multilabeled trees. *ACM Transactions on Algorithms*, 7(4):article 52, 2011.

[9] J. Barbay and G. Navarro. On compressing permutations and adaptive sorting. *Theoretical Computer Science*, 513:109–123, 2013.

[10] D. Belazzougui and G. Navarro. New lower and upper bounds for representing sequences. In *Proc. 20th Annual European Symposium on Algorithms (ESA)*, LNCS 7501, pages 181–192, 2012.

[11] T. Beller, S. Gog, E. Ohlebusch, and T. Schnattinger. Computing the longest common prefix array based on the Burrows-Wheeler transform. In *Proc 18th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 7024, pages 197–208, 2011.

[12] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[13] 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 International Symposium on Algorithms and Data Structures (WADS)*, LNCS 5664, pages 98–109, 2009.

[14] N. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Implicit indexing of natural language text by reorganizing bytecodes. *Information Retrieval*, 15(6):527–557, 2012.

[15] N. Brisaboa, M. Luaces, G. Navarro, and D. Seco. A fun application of compact data structures to indexing geographic data. In *Proc. 5th International Conference on Fun with Algorithms (FUN)*, pages 77–88, 2010.

[16] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation, 1994.

[17] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.

[18] Y.-F. Chien, W.-K. Hon, R. Shah, and J. Vitter. Geometric Burrows-Wheeler transform: Linking range searching and text indexing. In *Proc. 18th Data Compression Conference (DCC)*, pages 252–261, 2008.

[19] D. Clark. *Compact Pat Trees*. PhD thesis, Univ. of Waterloo, Canada, 1996.

[20] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5280, pages 176–187, 2008.

[21] F. Claude and G. Navarro. Extended compact Web graph representations. In *Algorithms and Applications (Ukkonen Festschrift)*, LNCS 6060, pages 77–91, 2010.

[22] F. Claude and G. Navarro. Self-indexed grammar-based compression. *Fundamenta Informaticae*, 111(3):313–337, 2010.

[23] F. Claude and G. Navarro. The wavelet matrix. In *Proc. 19th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 7608, pages 167–179, 2012.

[24] F. Claude, P. Nicholson, and D. Seco. Space efficient wavelet tree construction. In *Proc. 18th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 185–196, 2011.

[25] T. Cover and J. Thomas. *Elements of Information Theory.* Wiley, 1991.

[26] P. Ferragina, R. Giancarlo, and G. Manzini. The myriad virtues of wavelet trees. *Information and Computation*, 207(8):849–866, 2009.

[27] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *Journal of the ACM*, 57(1):article 4, 2009.

[28] P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.

[29] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):article 20, 2007.

[30] T. Gagie, J. Kärkkäinen, G. Navarro, and S.J. Puglisi. Colored range queries and document retrieval. *Theoretical Computer Science*, 483:36–50, 2013.

[31] T. Gagie, G. Navarro, and S. J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science*, 426-427:25–41, 2012.

[32] T. Gagie, S. J. Puglisi, and A. Turpin. Range quantile queries: Another virtue of wavelet trees. In *Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5721, pages 1–6, 2009.

[33] A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373, 2006.

[34] R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Proc. 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, 2005. Posters.

[35] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[36] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2006.

[37] R. Grossi, J. Vitter, and B. Xu. Wavelet trees: From theory to practice. In *Proc. 1st International Conference on Data Compression, Communications and Processing (CCP)*, pages 210–221, 2011.

[38] M. He and I. Munro. Succinct representations of dynamic strings. In *Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 334–346, 2010.

[39] D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the I.R.E.*, 40(9):1090–1101, 1952.

[40] G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[41] J. Kärkkäinen and S. J. Puglisi. Fixed block compression boosting in FM-indexes. In *Proc. 18th SPIRE*, pages 174–184, 2011.

[42] S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

[43] V. Mäkinen and G. Navarro. Position-restricted substring searching. In *Proc. 7th Latin American Symposium on Theoretical Informatics (LATIN)*, pages 703–714, 2006.

[44] V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *Proc. 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 4726, pages 214–226, 2007.

[45] V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007.

[46] V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms*, 4(3):article 32, 2008.

[47] C. Makris. Wavelet trees: a survey. *Computer Science and Information Systems*, 9(2):585–625, 2012.

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

[49] I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 1180, pages 37–42, 1996.

[50] G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms*, 2(1):87–114, 2004.

[51] G. Navarro. Wavelet trees for all. In *Proc. 23rd Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 7354, pages 2–26, 2012.

[52] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

[53] G. Navarro and Y. Nekrich. Top-k document retrieval in optimal time and linear space. In *Proc. 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1066–1078, 2012.

[54] G. Navarro and Y. Nekrich. Optimal dynamic sequence representations. In *Proc. 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 865–876, 2013.

[55] G. Navarro, Y. Nekrich, and L. Russo. Space-efficient data-analysis queries on grids. *Theoretical Computer Science*, 482:60–72, 2013.

[56] G. Navarro and A. Ordóñez. Compressing Huffman models on large alphabets. In *Proc. 23rd Data Compression Conference (DCC)*, pages 381–390, 2013.

[57] G. Navarro and E. Providel. Fast, small, simple rank/select on bitmaps. In *Proc. 11th International Symposium on Experimental Algorithms (SEA)*, LNCS 7276, pages 295–306, 2012.

[58] G. Navarro and S. J. Puglisi. Dual-sorted inverted lists. In *Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 310–322, 2010.

[59] G. Navarro and K. Sadakane. Fully-functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3):article 16, 2014.

[60] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th ALENEX*, pages 60–70, 2007.

[61] R. Raman, V. 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.

[62] D. Salomon. *Data Compression*. Springer, 2007.

[63] T. Schnattinger, E. Ohlebusch, and S. Gog. Bidirectional search in a string with wavelet trees. In *Proc. 21st Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 6129, pages 40–50, 2010.

[64] E. Schwartz and B. Kallick. Generating a canonical prefix encoding. *Communications of the ACM*, 7(3):166–169, 1964.

[65] G. Tischler. On wavelet tree construction. In *Proc. 22nd Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 208–218, 2011.

[66] N. Välimäki and V. Mäkinen. Space-efficient algorithms for document retrieval. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 205–215, 2007.

[67] S. Vigna. Broadword implementation of rank/select queries. In *Proc. 7th International Workshop on Experimental Algorithms (WEA)*, LNCS 5038, pages 154–168, 2008.

[68] Y. Zhang, Z. Pei, J. Yang, and Y. Liang. Canonical Huffman code based full-text index. *Progress in Natural Science*, 18(3):325–330, 2008.