The Wavelet Matrix

Francisco Claude[1,⋆] and Gonzalo Navarro[2,⋆⋆]

[1] David R. Cheriton School of Computer Science, University of Waterloo.
[2] Department of Computer Science, University of Chile.

Abstract. The *wavelet tree* (Grossi et al., SODA 2003) is nowadays a popular succinct data structure for text indexes, discrete grids, and many other applications. When it has many nodes, a levelwise representation proposed by Mäkinen and Navarro (LATIN 2006) is preferable. We propose a different arrangement of the levelwise data, so that the bitmaps are shuffled in a different way. The result can no more be called a wavelet tree, and we dub it *wavelet matrix*. We demonstrate that the wavelet matrix is simpler to build, simpler to query, and faster in practice than the levelwise wavelet tree. This has a direct impact on many applications that use the levelwise wavelet tree for different purposes.

1 Introduction

The *wavelet tree* [20] is a data structure designed to represent a sequence $S[1,n]$ over alphabet $[0,\sigma)$ and answer some queries on it. The following queries are sufficient to provide efficient data structures for many applications:

- $\texttt{access}(S,i)$ 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.

A wavelet tree is a balanced binary tree with σ leaves and $\sigma-1$ internal nodes, each of which holds a bitmap. In its most basic form, the bitmaps add up to $n\lceil\lg\sigma\rceil$ bits. Those bitmaps are equipped with sublinear-size structures to carry out binary `rank` and `select` operations. Considering carefully implemented pointers of $\lg n$ bits for the tree, the basic wavelet tree requires $n\lg\sigma + o(n\lg\sigma) + O(\sigma\lg n)$ bits. This is asymptotically equivalent to a plain representation of S, yet the wavelet tree is able to solve the three operations in time $O(\lg\sigma)$. However, in applications where the alphabet is large, the $O(\sigma\lg n)$ term may become dominant (both in theory and in practice). Mäkinen and Navarro [24, 26] showed that it is possible to concatenate all the bitmaps of each level and still simulate the tree navigation using `rank` and `select` operations on the concatenated bitmaps. The size was reduced to $n\lg\sigma + o(n\lg\sigma)$ bits. While in theory the complexities stayed the same, in practice one needs three times the

⋆ Funded by Google U.S./Canada PhD Fellowship.
⋆⋆ Funded in part by Millennium Nucleus Information and Coordination in Networks ICM/FIC P10-024F, Chile.

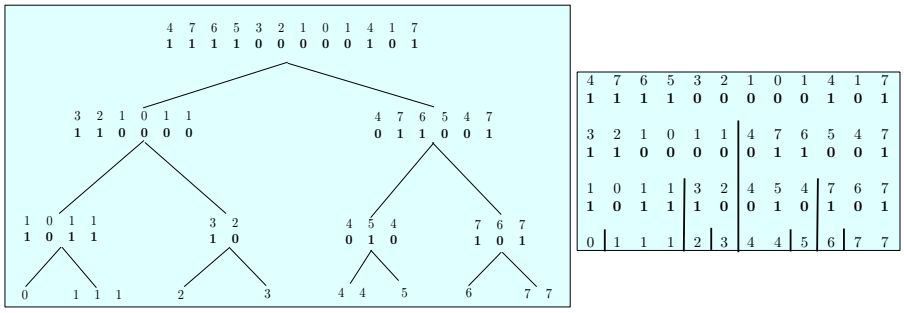

Fig. 1. On the left, the standard wavelet tree over a sequence. The subsequences S_v are not stored. The bitmaps B_v, in bold, are stored, as well as the tree topology. On the right, its levelwise version. The divisions into nodes are not stored but computed on the fly.

number of operations carried out over a standard wavelet tree. This slowdown has been accepted since then as an unavoidable price to pay for having the tree structure in implicit form.

In this paper we show that insisting in simulating the original wavelet tree was not the best idea. We introduce a different levelwise arrangement of the bits that turns out to simplify all the operations. The result recovers much of the performance of the original wavelet tree, and offers the same functionality. The structure cannot properly be called a "tree"; rather, we call it a *wavelet matrix*.

Our result, which is of practical nature, will have a large impact in a number of applications where the levelwise wavelet tree was used: compressed full-text indexes [22, 29, 14, 24, 7, 4, 11, 23], inverted indexes [9, 1, 31, 17], document retrieval [36, 18, 16, 13, 17, 32], graph representations [9, 10], discrete grids [4, 5, 33, 30], binary relations [2, 3], and general problems on numeric sequences [18, 17, 23]. All of those will become up to twice as fast by using wavelet matrices.

2 The Wavelet Tree

A wavelet tree [20] 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, it stores 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. 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, exactly n bits per level (the lowest one may hold fewer bits). Thus it stores at

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

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

Therefore, the 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 [21, 28, 8].

The process to compute $\texttt{rank}_a(S, i)$ 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: 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 from one level to the other, we update i to be the number of times the current bit of a appears up to position i in the node. When we arrive at the leaf handling the range $[a, a]$, the answer to `rank` is the value of i at that leaf.

Finally, to compute $\texttt{select}_a(S, j)$ 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 $j_v = \texttt{select}_0(B_v, j_u)$. Else, the corresponding position is $j_v = \texttt{select}_1(B_v, j_u)$. When we finally arrive at the root ν, the answer to the query is j_ν.

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

3 The Levelwise Wavelet Tree

Since the wavelet tree is a complete balanced binary tree, it is possible to concatenate all the bitmaps at each level and still retain the same functionality [24, 26]. 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 complication is how to keep track of the range $\tilde{B}_\ell[s_v, e_v]$ corresponding to a node v of depth ℓ.

3.1 The Strict Variant

The strict variant [24, 26] 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

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 $\texttt{rank}_a(S, i)$; and $\mathbf{sel}(\nu, a, j)$ returns $\texttt{select}_a(S, j)$. The left/right children of v are called v_l/v_r.

```
acc(v, i)
  if ω_v − α_v = 1 then
    return α_v
  end if
  if B_v[i] = 0 then
    i ← rank_0(B_v, i)
    return acc(v_l, i)
  else
    i ← rank_1(B_v, i)
    return acc(v_r, i)
  end if
```

```
rnk(v, a, i)
  if ω_v − α_v = 1 then
    return i
  end if
  if a < 2^⌈lg(ω_v − α_v)⌉−1
  then
    i ← rank_0(B_v, i)
    return rnk(v_l, a, i)
  else
    i ← rank_1(B_v, i)
    return rnk(v_r, a, i)
  end if
```

```
sel(v, a, j)
  if ω_v − α_v = 1 then
    return j
  end if
  if a < 2^⌈lg(ω_v − α_v)⌉−1
  then
    j ← sel(v_l, a, j)
    return select_0(B_v, j)
  else
    j ← sel(v_r, a, j)
    return select_1(B_v, j)
  end if
```

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`). Thus the times are expected to triple for these queries. For query `select`, the strict variant requires two extra binary `rank` operations per original binary `select`. Since in practice the binary `select` is more expensive than `rank`, the impact on query `select` is lower.

3.2 The Extended Variant

The *extended* variant [9], 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 [34]), but the constant is much lower than on a pointer-based tree (which stores a left child, right child, a parent pointer, the value n_v, the pointer to bitmap B_v, an equivalent to array C, etc.).

With the help of array C, the number of operations equals that of the standard version, since array C lets us compute the ranges: The range of any node v is simply $[C[\alpha_v] + 1, C[\omega_v]]$. In the algorithms for queries `access` and `rank`, where we descend from the root, the values α_v and ω_v are easily maintained.

Algorithm 2 Levelwise 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 $\texttt{rank}_a(S, i)$; and $\mathbf{sel}(0, a, j, 0, n)$ returns $\texttt{select}_a(S, j)$. For simplicity we have omitted the computation of $[\alpha_v, \omega_v)$.

```
acc(ℓ, i, p, e)
  if ω_v − α_v = 1 then
    return α_v
  end if
  l ← rank_0(B̃_ℓ, p)
  r ← rank_0(B̃_ℓ, e)
  if B̃_ℓ[i] = 0 then
    z ← rank_0(B̃_ℓ, p + i)
    return acc(ℓ+1,
      z−l, p, p+r−l)
  else
    z ← rank_1(B̃_ℓ, p + i)
    return acc(ℓ+1,
      z−(p−l), p+r−l, e)
  end if
```

```
rnk(ℓ, a, i, p, e)
  if ω_v − α_v = 1 then
    return i
  end if
  l ← rank_0(B̃_ℓ, p)
  r ← rank_0(B̃_ℓ, e)
  if a < 2^⌈lg(ω_v − α_v)⌉−1 then
    z ← rank_0(B̃_ℓ, p + i)
    return rnk(ℓ+1, a,
      z−l, p, p+r−l)
  else
    z ← rank_1(B̃_ℓ, p + i)
    return rnk(ℓ+1, a,
      z−(p−l), p+r−l, e)
  end if
```

```
sel(ℓ, a, j, p, e)
  if ω_v − α_v = 1 then
    return j
  end if
  l ← rank_0(B̃_ℓ, p)
  r ← rank_0(B̃_ℓ, e)
  if a < 2^⌈lg(ω_v − α_v)⌉−1 then
    j←sel(ℓ+1, a, j, p, p+r−l)
    return
      select_0(B̃_ℓ, l + j)−p
  else
    j←sel(ℓ+1, a, j, p+r−l, e)
    return
      select_1(B̃_ℓ, (p−l)+j)−p
  end if
```

This is slightly more complicated when solving query $\texttt{select}_a(S, j)$. We start at offset $C[a] + j$ and track this position upwards: If the leaf is a left child of its parent (i.e., if a is even), then the parent's range (in the deepest bitmap $\tilde{B}_\ell$) is $[C[a] + 1, C[a + 2]]$. Instead, if the leaf is a right child of its parent, then the parent's range is $[C[a - 1] + 1, C[a + 1]]$. We use binary `select` on this range to map the position j to the parent's range. Now we proceed similarly at the parent, from range $[C[a'] + 1, C[a' + 2]]$ (where $a' = a$ or $a - 1$ is even). If $a' = 0 \bmod 4$, then this node is a left child, otherwise it is a right child. In the first case, it corresponds to range $[C[a'] + 1, C[a' + 4]]$ in bitmap $\tilde{B}_{\ell-1}$, otherwise it is $[C[a' - 2] + 1, C[a' + 2]]$. We continue until the root, where j is the answer.

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 unnecessary 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 strict levelwise 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 $\texttt{rank}_0(\tilde{B}_\ell, i)$. If $\tilde{B}_\ell[i] = 1$, the position at level $\ell + 1$ will be $z_\ell + \texttt{rank}_1(\tilde{B}_\ell, i)$. 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

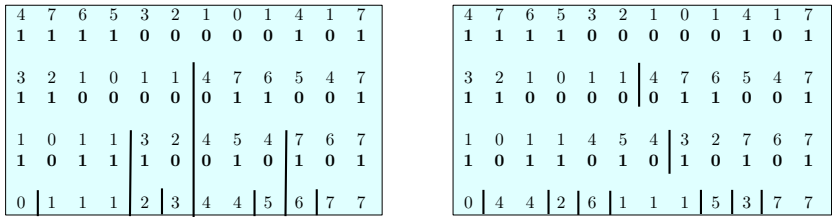

Fig. 2. On the left, the levelwise wavelet tree of the previous example (Figure 1). On the right, the wavelet matrix over the same sequence. One vertical line per level represents the position stored in the z_ℓ values.

range in $\tilde{B}_\ell$. This is obviously true for the root ν. Now, assuming it is true for v, with interval $\tilde{B}_\ell[s_v, e_v]$, all the positions with $\tilde{B}_\ell[i] = 0$ for $s_v \leq i \leq e_v$ 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 2 (left) illustrates the wavelet matrix, where it can be seen that the blocks of the wavelet tree are maintained, albeit in different order.

We describe now how to carry out $\mathtt{access}(S, i)$. If $\tilde{B}_0[i] = 0$, we set i to $\mathtt{rank}_0(\tilde{B}_0, i)$. Else we set i to $z_0 + \mathtt{rank}_1(\tilde{B}_0, i)$. Now we descend to level 1, and continue until reaching a leaf. The sequence of bits $\tilde{B}_\ell[i]$ 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 $\mathtt{rank}_a(S, i)$. This time we need to keep track of the position i, and also of the position preceding the range, initially $p = 0$. 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 to $\mathtt{rank}_0(\tilde{B}_\ell, p)$ and i to $\mathtt{rank}_0(\tilde{B}_\ell, i)$. Otherwise, we go "right" by mapping p to $z_\ell + \mathtt{rank}_1(\tilde{B}_\ell, p)$ and i to $z_\ell + \mathtt{rank}_1(\tilde{B}_\ell, i)$. When we arrive at the leaf level, the answer is $i - p$. 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) levelwise variant.

Finally, consider operation $\mathtt{select}_a(S, j)$. We first descend towards the leaf of a just as done for $\mathtt{rank}_a(S, i)$, keeping track only of p. When we arrive at the last level, p precedes the range corresponding to the leaf of a, and thus we wish to track upwards position $p + j$. The upward tracking of a position $\tilde{B}_{\ell+1}[i]$ is simple: If we went left from level ℓ, then this position was mapped from a 0 in $\tilde{B}_\ell$, and therefore it came from $\tilde{B}_\ell[\mathtt{select}_0(\tilde{B}_\ell, i)]$. Otherwise, position i was mapped from a 1, and thus it came from $\tilde{B}_\ell[\mathtt{select}_1(\tilde{B}_\ell, i - z_\ell)]$. When we arrive at the root bitmap, i 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 the overhead is low.

Algorithm 3 gives the pseudocode.

Construction. Construction of the wavelet matrix is even simpler than that of the levelwise wavelet tree, because we do not need to care for node boundaries. At the first level we keep in bitmap $\tilde{B}_0$ the highest bits of the symbols in S, and

Algorithm 3 Wavelet matrix algorithms: On the wavelet matrix of sequence S, $\mathbf{acc}(0,i)$ returns $S[i]$; $\mathbf{rnk}(0,a,i,0)$ returns $\texttt{rank}_a(S,i)$; and $\mathbf{sel}(0,a,j,0)$ returns $\texttt{select}_a(S,j)$. For simplicity we have omitted the computation of $[\alpha_v,\omega_v)$.

```
acc(ℓ, i)
  if ω_v − α_v = 1 then
    return α_v
  end if
  if B̃_ℓ[i] = 0 then
    i ← rank_0(B̃_ℓ, i)
  else
    i ← rank_1(B̃_ℓ, i)
  end if
  return acc(ℓ+1, i)
```

```
rnk(ℓ, a, i, p)
  if ω_v − α_v = 1 then
    return i − p
  end if
  if a < 2^⌈lg(ω_v − α_v)⌉−1 then
    p ← rank_0(B̃_ℓ, p)
    i ← rank_0(B̃_ℓ, i)
  else
    p ← z_ℓ + rank_1(B̃_ℓ, p)
    i ← z_ℓ + rank_1(B̃_ℓ, i)
  end if
  return rnk(ℓ+1, a, i, p)
```

```
sel(ℓ, a, j, p)
  if ω_v − α_v = 1 then
    return p + j
  end if
  if a < 2^⌈lg(ω_v − α_v)⌉−1 then
    p ← rank_0(B̃_ℓ, p)
    j ← sel(ℓ+1, a, j, p)
    return select_0(B̃_ℓ, j)
  else
    p ← z_ℓ + rank_1(B̃_ℓ, p)
    j ← sel(ℓ+1, a, j, p)
    return select_1(B̃_ℓ, j−z_ℓ)
  end if
```

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.

Compression. As in previous work, we can replace the plain representations of bitmaps $\tilde{B}_\ell$ by compressed ones [34], so that the total space is $nH_0(S) + o(n \lg \sigma)$ bits[20]. The concatenation of the bitmaps adds an extra space (positive or negative) that can be upper bounded by $O(\sigma \lg n)$, but is negligible in practice.

Practical considerations A problem that arises when combining the wavelet matrix with the $C[]$ array of the extended version is that the leaves are not in order. While this is easily fixed by reversing the bits of the symbols, this creates holes in C if σ is not a power of 2, even if the alphabet was originally contiguous (e.g., consider alphabet $0,1,2,3,4 = 000..100$; after reversing the bits we obtain positions $0,1,2,3,6$, so we need to allocate 7 cells instead of 5). This can make the size of C to double in the worst case.

5 Experimental Results

Our practical implementation is included in LIBCDS, a library implementing several space-efficient data structures, `http://libcds.recoded.cl`, version 1.0.12. For each wavelet tree/matrix variant we present two versions, `RG` and `RRR`. The first one corresponds to the implementation [19] of the proposals by Jacobson [21], and Clark [8] and Munro [28]. The second version, `RRR`, corresponds to the implementation [9] of the proposal of Raman, Raman and Rao [34].

The variants measured are `WT`: standard pointer-based wavelet tree; `WTNP`: the extended levelwise wavelet tree (i.e., No Pointers); and `WM`: the (extended) wavelet matrix (array C is used to perform `select` in a single upward traversal).

These names are composed with the bitmap implementations by appending the bitmap representation name. For example, we call `WTRRR` the standard pointer-based wavelet tree with all bitmaps represented with Raman, Raman and Rao's compressed bitmaps. For readability, we show only the extended versions. In general, they achieve space very close to the strict versions and yield better time performance.

Datasets We use four different datasets, left at `http://indexing.recoded.cl`:

- `ESWiki`: Sequence of word identifiers generated by stemming the Spanish Wikipedia (`http://es.wikipedia.org` dated 03/02/2010) with the Snowball algorithm. The sequence has length $n = 511{,}173{,}618$ and alphabet size $\sigma = 3{,}210{,}671$. This allows, say, simulating a positional inverted index [9, 1].
- `BWT`: The Burrows-Wheeler transform (BWT) [6] of `ESWiki`. The length and size of the alphabet match those of `ESWiki`. This is useful to implement many full-text compressed self-indexes [14, 15],
- `Indochina`: The concatenation of all adjacency lists of Web graph `Indochina-2004`, available at `http://law.dsi.unimi.it`. The length of the sequence is $n = 194{,}109{,}311$ and the alphabet size is $\sigma = 7{,}414{,}866$. This supports forward and backward traversals on the graph [9, 10].
- `INV`: Concatenation of inverted lists for a random sample of 2,961,510 documents from the English Wikipedia (`http://en.wikipedia.org`). This sequence has length $n = 338{,}027{,}430$ and its alphabet size is $\sigma = 2{,}961{,}510$. This is useful to simulate document inverted indexes [31, 17].

Measurements To measure performance we generated 100,000 inputs for each query and averaged their execution time. The `access` queries were generated by picking positions in the text uniformly at random. The `rank` queries were generated the same way as for access, associating to each position one symbol uniformly at random. Each `select` query was generated by first picking a symbol s uniformly at random, and then picking the positional argument for `select` uniformly at random from the range $[1, \texttt{rank}_s(S, n-1)]$.

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.

Results Figure 3 shows some of the results obtained for our four datasets (the rest are similar). As expected, the wavelet matrix improves upon the levelwise wavelet tree considerably, doubling (or more) the speed for `access` and improving (albeit less) on `rank` and `select`.

We also note that this new implementation of wavelet trees is competitive with the pointer-based wavelet trees for `access` and `rank` operations, which use much more space. For `select`, the pointer-based wavelet tree is faster. This is

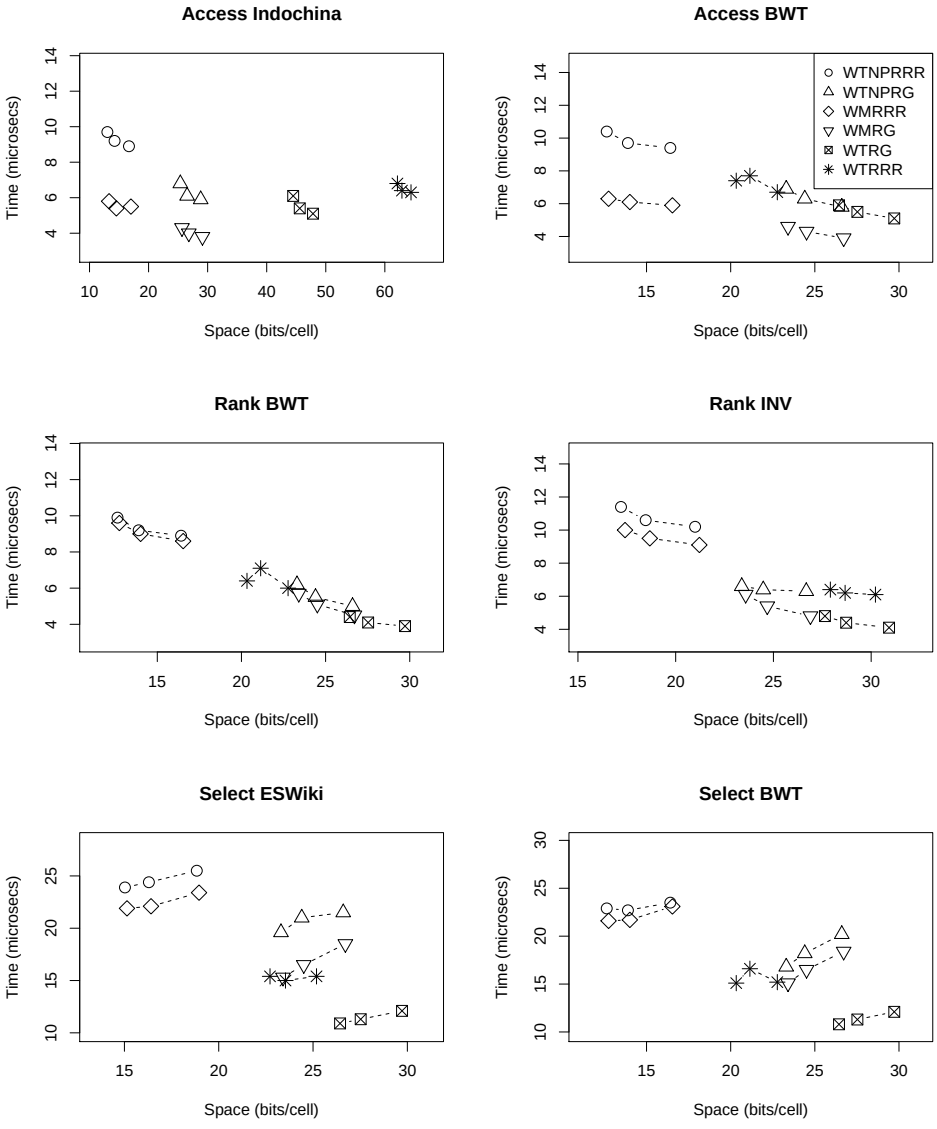

Fig. 3. Running time per query for rank, select and access over four datasets.

because the pointer-based wavelet tree performs `select` queries over smaller bitmaps, which in practice take time logarithmic on the bitmap length. This can be overcome by implementing a position-restricted version of binary `select`.

As for space, it can be seen that the wavelet matrix achieves virtually the same space as the levelwise wavelet tree, as expected. Pointer-based versions, instead, pose a noticeable overhead related to the alphabet size.

It is tempting to consider an alternative wavelet matrix, which emulates a radix sort of the sequence (i.e., it stably sorts first by the least significant bit and ends with the most significant bit). While this seems to be an innocent change with the advantage of leaving the leaves in sorted order, it does not preserve the wavelet tree blocks (even in scrambled form). Our experiments with this variant showed a behavior very similar to the original in `ESWiki`, despite not preserving blocks. Even in the case of `BWT`, which has long runs of the same symbol, those runs are preserved even if the blocks are destroyed. Hence `RRR` compression is not affected. The case of `Indochina` and `INV`, however, was different. These are formed by long substrings of increasing values with small differences, which induce long runs in the bitmaps in a decomposition by highest-bit-first. However, such runs do not appear in a lowest-bit-first decomposition. As a result, the space with `RRR` compression was much worse than on our original variant.

6 Conclusions

The (strict) levelwise wavelet tree [24, 26], designed to avoid the $O(\sigma \lg n)$ space overhead of standard wavelet trees [20], 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 strict levelwise wavelet trees. It requires $n \lg \sigma + o(n \lg \sigma)$ bits of space, it can be built in $O(n \lg \sigma)$ time and almost in-place.

There are many more sophisticated aspects of wavelet trees, which we have ignored in this paper for simplicity. We briefly sketch them here:

Range searches: Levelwise wavelet trees are particularly useful for representing discrete $n \times n$ grids, where $\sigma = n$. They use algorithms that are slightly more complex than our `access`/`rank`/`select`, for example they track ranges downwards, usually to the left *and* to the right of the current node. All those algorithms can perfectly be executed over the wavelet matrix. The fact that the nodes at each level are scrambled is immaterial to the algorithms.

Dynamization: Inserting and deleting symbols of S can be carried out without any complication, by tracking the position to insert (just like `rank`) and to delete (just like `access`), and therefore all the results on dynamic wavelet trees [27] translate directly to wavelet matrices.

Construction: We have built the wavelet matrix within n bits of extra space. There are even more space-efficient constructions for wavelet trees [12, 35]. It would be interesting to find out whether they apply to wavelet matrices.

Multiary: Multiary wavelet trees [15] can also be adapted 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, $O(\lg^2 n \lg \sigma)$.

Implicit compression boosting: Mäkinen and Navarro [25, 27] proved that the wavelet tree of the BWT [6] of a text T, if its bitmaps are compressed to zero-order entropy (e.g., using Raman et al. [34]), would achieve high-order entropy compression of T. This was essential to simplify compressed text indexing and to enable dynamic variants. Their results apply to wavelet matrices as well, because all that is required is that the nodes are contiguous in the levelwise bitmaps B_ℓ, being irrelevant which is the relative order of the nodes. This effect can already be noticed in our experiments; compare the spaces on `EsWiki` with its BWT version `BWT`.

Acknowledgement. Thanks to Daisuke Okanohara for useful comments.

References

1. Arroyuelo, D., González, S., Oyarzún, M.: Compressed self-indices supporting conjunctive queries on document collections. In: Proc. 17th SPIRE. pp. 43–54 (2010)
2. Barbay, J., Claude, F., Navarro, G.: Compact rich-functional binary relation representations. In: Proc. 9th LATIN. pp. 170–183 (2010)
3. Barbay, J., Claude, F., Navarro, G.: Compact binary relation representations with rich functionality. CoRR abs/1201.3602 (2012)
4. Bose, P., He, M., Maheshwari, A., Morin, P.: Succinct orthogonal range search structures on a grid with applications to text indexing. In: Proc. 11th WADS. pp. 98–109 (2009)
5. Brisaboa, N., Luaces, M., Navarro, G., Seco, D.: A fun application of compact data structures to indexing geographic data. In: Proc. 5th FUN. pp. 77–88 (2010)
6. Burrows, M., Wheeler, D.: A block sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation (1994)
7. Chien, Y.F., Hon, W.K., Shah, R., Vitter, J.: Geometric Burrows-Wheeler transform: Linking range searching and text indexing. In: Proc. 18th DCC. pp. 252–261 (2008)
8. Clark, D.: Compact Pat Trees. Ph.D. thesis, Univ. of Waterloo, Canada (1996)
9. Claude, F., Navarro, G.: Practical rank/select queries over arbitrary sequences. In: Proc. 15th SPIRE. pp. 176–187 (2008)
10. Claude, F., Navarro, G.: Extended compact Web graph representations. In: Algorithms and Applications (Ukkonen Festschrift). pp. 77–91 (2010)
11. Claude, F., Navarro, G.: Self-indexed grammar-based compression. Fund. Inf. 111(3), 313–337 (2010)
12. Claude, F., Nicholson, P., Seco, D.: Space efficient wavelet tree construction. In: Proc. 18th SPIRE. pp. 185–196 (2011)
13. Culpepper, J.S., Navarro, G., Puglisi, S.J., Turpin, A.: Top-k ranked document search in general text databases. In: Proc. 18th ESA. pp. 194–205 (part II) (2010)
14. Ferragina, P., Manzini, G.: Indexing compressed texts. J. ACM 52(4), 552–581 (2005)
15. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Trans. Alg. 3(2), article 20 (2007)

16. Gagie, T., Navarro, G., Puglisi, S.J.: Colored range queries and document retrieval. In: Proc. 17th SPIRE. pp. 67–81 (2010)
17. Gagie, T., Navarro, G., Puglisi, S.J.: New algorithms on wavelet trees and applications to information retrieval. Theor. Comp. Sci. 426-427, 25–41 (2012)
18. Gagie, T., Puglisi, S.J., Turpin, A.: Range quantile queries: Another virtue of wavelet trees. In: Proc. 16th SPIRE. pp. 1–6 (2009)
19. González, R., Grabowski, S., Mäkinen, V., Navarro, G.: Practical implementation of rank and select queries. In: Proc. 4th Workshop on Efficient and Experimental Algorithms (WEA). pp. 27–38 (2005), posters
20. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: Proc. 14th SODA. pp. 841–850 (2003)
21. Jacobson, G.: Space-efficient static trees and graphs. In: Proc. 30th FOCS. pp. 549–554 (1989)
22. Kärkkäinen, J.: Repetition-Based Text Indexing. Ph.D. thesis, Univ. of Helsinki, Finland (1999)
23. Kreft, S., Navarro, G.: Self-indexing based on LZ77. In: Proc. 22nd CPM. pp. 41–54 (2011)
24. Mäkinen, V., Navarro, G.: Position-restricted substring searching. In: Proc. 7th LATIN. pp. 703–714 (2006)
25. Mäkinen, V., Navarro, G.: Implicit compression boosting with applications to self-indexing. In: Proc. 14th SPIRE. pp. 214–226 (2007)
26. Mäkinen, V., Navarro, G.: Rank and select revisited and extended. Theor. Comp. Sci. 387(3), 332–347 (2007)
27. Mäkinen, V., Navarro, G.: Dynamic entropy-compressed sequences and full-text indexes. ACM Trans. Alg. 4(3), article 32 (2008)
28. Munro, I.: Tables. In: Proc. 16th FSTTCS. pp. 37–42 (1996)
29. Navarro, G.: Indexing text using the Ziv-Lempel trie. J. Discr. Alg. 2(1), 87–114 (2004)
30. Navarro, G., Nekrich, Y., Russo, L.: Space-efficient data-analysis queries on grids. CoRR abs/1106.4649v2 (2012)
31. Navarro, G., Puglisi, S.J.: Dual-sorted inverted lists. In: Proc. 17th SPIRE. pp. 310–322 (2010)
32. Navarro, G., Puglisi, S.J., Valenzuela, D.: Practical compressed document retrieval. In: Proc. 10th SEA. pp. 193–205 (2011)
33. Navarro, G., Russo, L.: Space-efficient data-analysis queries on grids. In: Proc. 22nd ISAAC. pp. 323–332 (2011)
34. Raman, R., Raman, V., Rao, S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proc. 13th SODA. pp. 233–242 (2002)
35. Tischler, G.: On wavelet tree construction. In: Proc. 22nd CPM. pp. 208–218 (2011)
36. Välimäki, N., Mäkinen, V.: Space-efficient algorithms for document retrieval. In: Proc. 18th CPM. pp. 205–215 (2007)