Online Construction of Wavelet Trees*

Paulo G. S. da Fonseca[1] and Israel B. F. da Silva[2]

1 **Centro de Informática, Universidade Federal de Pernambuco, Recife, Brazil**
`paguso@cin.ufpe.br`
2 **Centro de Informática, Universidade Federal de Pernambuco, Recife, Brazil**
`ibfs@cin.ufpe.br`

Abstract

The wavelet tree (WT) is a flexible and efficient data structure for representing character strings in succinct space, while allowing for fast generalised rank, select and access operations. As such, they play an important role in modern text indexing methods. However, despite their popularity, not many algorithms have been published concerning their construction. In particular, while the WT is capable of representing a sequence of length n over an alphabet of size m in $n \lg m + o(n \lg m)$ bits, much more space is typically used for its construction. Here we propose an $O(n \lg m)$-time *online* method for the construction of the WT, requiring no prior knowledge about the input alphabet. The proposed algorithm is conceptually simpler than other state-of-the-art methods, while having comparable time performance and being more space-efficient in practice, since it performs just one pass over the input text and uses little extra space other than for the structure itself, as shown both theoretically and empirically.

1998 ACM Subject Classification E.1 Data Structures

Keywords and phrases Wavelet tree, Online construction

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.16

1 Introduction

The wavelet tree is a fundamental data structure proposed by Grossi, Gupta and Vitter [10], whose myriad virtues have been widely recognised since its introduction [6, 18, 17]. They are used for representing large character strings in tight space, while allowing for some specific questions about their composition, for instance 'how many occurrences of a particular letter are there between positions 3- to 8-billion?', to be answered very quickly, without having to probe the string. It is thus a cornerstone of modern string matching techniques.

Despite having been introduced for well over a decade, not many papers have been published that deal specifically with their construction. In this paper we present a simple and efficient method for the online construction of wavelet trees. We contend that the algorithm proposed herein is among the most efficient methods in practice, both in time and space, and provide experimental evidence to support this claim.

We begin by describing the data structure, its main operations and space requirements. Then we present our construction algorithm with a theoretical analysis of its costs. After that we discuss previous related work, and where our contribution sits in the context. Finally, we report on our experimental analysis and state our conclusions.

* This work was supported by the brazilian Conselho Nacional de Pesquisa – CNPq (Project MCTI/CNPq/Universal 449842/2014-2) and the Fundação de Amparo à Ciência e Tecnologia de Pernambuco – FACEPE (Project APQ-0587-1.03/14).

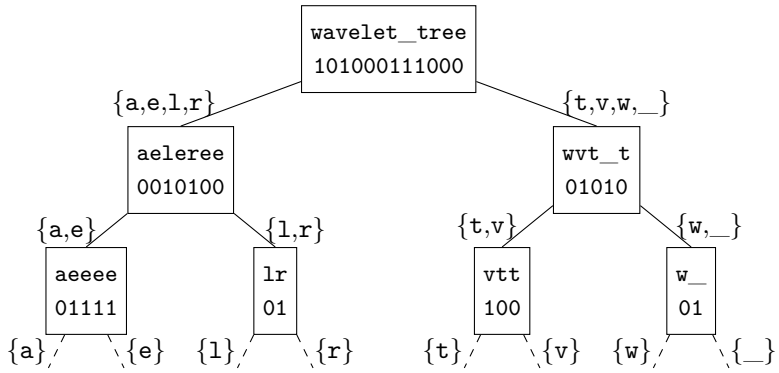

Figure 1 Balanced WT of $T = \texttt{wavelet\_tree}$ over $\mathcal{A} = \{\texttt{a}, \texttt{e}, \texttt{l}, \texttt{r}, \texttt{t}, \texttt{v}, \texttt{w}, \texttt{\_}\}$. Only the bit vectors are actually stored. The strings and alphabets are shown for illustration purposes only.

2 The Wavelet Tree

We consider strings $T = t_0 \cdots t_{n-1}$ over a finite alphabet $\mathcal{A} = \{a_0, \ldots, a_{m-1}\}$. For any subset $\mathcal{A}' \subseteq \mathcal{A}$, let $\mathrm{proj}(T, \mathcal{A}')$ be the noncontiguous, possibly empty, subsequence of T consisting of all its positions in $\mathcal{A}'$. We call this the *projection* of T on $\mathcal{A}'$. In particular, $\mathrm{proj}(T, [l:r])$, with $0 \le l, r \le m$, denotes the subsequence of T made of its characters in the range $\mathcal{A}[l:r] \equiv \{a_l, \ldots, a_{r-1}\}$. For example, if $\mathcal{A} = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$ and $T = \texttt{adbcabdcb}$, then $\mathrm{proj}(T, [1:3]) = \texttt{bcbcb}$. Complementary, we define the *support* of T as $\mathrm{supp}(T) = \cup_{i=0}^{n-1}\{t_i\}$, that is, the subset of $\mathcal{A}$ consisting of the characters in T. Hence we have $\mathrm{proj}(T, \mathrm{supp}(T)) = T$, and $\mathrm{supp}(\mathrm{proj}(T, \mathcal{A}')) \subseteq \mathcal{A}'$.

▶ **Definition 1.** A wavelet tree (WT) of a nonempty string T with support $\mathcal{A}$ is a digital search tree recursively defined as

$$W(T, \mathcal{A}) = \begin{cases} \bot, & \text{if } |\mathcal{A}| = 1, \\ \begin{array}{c} B(T, \mathcal{A}_0, \mathcal{A}_1) \\ \diagup \quad \diagdown \\ W(\mathrm{proj}(T, \mathcal{A}_0), \mathcal{A}_0) \quad W(\mathrm{proj}(T, \mathcal{A}_1), \mathcal{A}_1) \end{array}, & \text{otherwise,} \end{cases} \tag{1}$$

where

- $\mathcal{A} = \mathcal{A}_0 \cup \mathcal{A}_1$ is a nontrivial partition of the alphabet,
- The root consists in an indicator bit array $B(T, \mathcal{A}_0, \mathcal{A}_1) = b_0 \cdots b_{n-1}$ of length $n = |T|$, such that $b_i = 0$, if $t_i \in \mathcal{A}_0$, or $b_i = 1$, if $t_i \in \mathcal{A}_1$,
- The left and right subtrees, $W(\mathrm{proj}(T, \mathcal{A}_0), \mathcal{A}_0)$ and $W(\mathrm{proj}(T, \mathcal{A}_1), \mathcal{A}_1)$ correspond to wavelet trees of the projections of T over the subalphabets $\mathcal{A}_0$ and $\mathcal{A}_1$, respectively, and
- $\bot$ represents a null (empty) tree, which is the base case for unary alphabets.

The most common case of the definition of $W(T, \mathcal{A})$ happens when, at each node, the alphabet is split into two halves, as shown in Figure 1. In this case, the WT is said to be *balanced*.

The WT data structure provides for generalised *access*, *rank*, and *select* queries. Given $W = W(T, \mathcal{A})$, $W.\mathrm{access}(i)$ returns the character t_i, $W.\mathrm{rank}(c, i)$ returns the number of occurrences of character c in T up to (and including) position i, and $W.\mathrm{select}(c, j)$ returns the position of the jth occurrence of character c in T, so that if $i = W.\mathrm{select}(c, j)$, then

$W.\text{rank}(c, i) = j$. In order to do so efficiently both in time and space, the WT employs specialised bit vectors supporting constant-time binary rank and select primitive operations, at the cost of only a sublinear amount of extra bits $E(r) = o(r)$, where r is the lenght of the raw sequence of bits [1, 11]. Hence the balanced WT requires $\approx \lg m(n+o(n)) = n \lg m + o(n \lg m)$ bits of space for the bit vectors since, at every level of the tree, each letter of the text is represented exactly once. On top of that, we have one pointer per node, each taking $O(\lg n)$ bits, thus $O(m \lg n)$ bits for the tree structure. This $O(m \lg n)$ factor can actually be avoided, as discussed in [18], and we can have a representation with just one bitvector and no pointers in $n \lg m + o(n \lg m)$ bits of space, which qualifies the WT as a *succinct* data structure [15], since the uncompressed sequence takes $n \lg m$ bits. Moreover, the generalised access, rank and select queries can be answered in $O(\lg m)$ time by following root-to-leaf paths and performing rank and select queries on their bit arrays.

3 WT construction

The naive recursive procedure for building $W(T, \mathcal{A})$ that mirrors Definition 1 requires partitioning $\mathcal{A}$, computing the indicator bit array of the root node, $B(T, \mathcal{A}_0, \mathcal{A}_1)$, based on each character of T belonging to either $\mathcal{A}_0$ or $\mathcal{A}_1$, and recursively building the left and right subtrees from the projections of T over these subalphabets. In the balanced WT, at each node subalphabets can be represented as $[l : r]$ pairs, and the bit array and the projections can be computed in one pass over the corresponding substring. In total, we have $O(n \lg m)$ time complexity. Nevertheless this strategy requires creating and maintaining several intermediate substrings (the projections) through the recursion stack, and going over multiple copies of the same character of T.

The main motivation of our method is to completely avoid the costly operations of explicitly partitioning the alphabet and projecting the strings, or any other expensive substring manipulation like character counting or sorting. We want to process the input string in one single pass and so our proposed algorithms are *online*, that is we scan each character of T exactly once from left to right, and maintain no extra copies.

For the sake of presentation only, we consider two different scenarios. In the first case, the support alphabet is known in advance. This situation is used as an introduction to the more general case where the alphabet is unknown, and revealed only as the string is scanned.

3.1 WT construction with known alphabet

The first procedure, shown in Algorithm 1, is used to build a WT for a string T whose support alphabet is given. The idea is very simple and consists in, first, initialising an empty WT, called *template*, and then filling the bit arrays in one scan of T. For every scanned character t_i, the algorithm follows the corresponding (unique) root-to-leaf path, appending the necessary bits to the nodes on its way.

There is just one subtlety that will prove important in what follows. We have been accustomed to depictions of balanced WTs where the alphabet is literally cut in half, with the first $\lceil m/2 \rceil$ symbols assigned to the first subalphabet, and the remaining $\lfloor m/2 \rfloor$ to the second one. This is equivalent to assigning a_j to left or right subtrees down the root based on the binary representation of j from left to right, that is, from the most to the least significant bit. However, this need not be the case in general, for it suffices that the partition be evenly sized so as to keep the tree balanced, irrespective of which symbol goes where. So, instead, we choose to follow the bits of j from right (lsb) to left (msb). This results in symbols being assigned to the left/right subalphabets in an alternate fashion.

Algorithm 1 Balanced WT online construction with known alphabet

```
1: Algorithm WT0 (T = t_0 ··· t_{n−1}, A = {a_0, ..., a_{m−1}})
2:     root ← buildTemplate([0 : m])
3:     for i ← 0, ..., n − 1 do
4:         cur ← root
5:         j ← A.index(t_i)
6:         while cur ≠ ⊥ do
7:             if j mod 2 = 0 then
8:                 cur.B.append(0)
9:                 cur ← cur.left
10:            else
11:                cur.B.append(1)
12:                cur ← cur.right
13:            j ← j ≫ 1                ▷ Right shift
14:    return root
```

```
1: Algorithm buildTemplate ([l : r])
2:     if (r − l) = 1 then
3:         return ⊥
4:     root ← new empty WT node
5:     h ← ⌈(l + r)/2⌉
6:     root.left ← buildTemplate([0 : h])
7:     root.right ← buildTemplate([h : m])
8:     return root
```

▶ **Proposition 2.** *Algorithm 1 builds $W(T, \mathcal{A})$ in $O(n \lg m)$ time, using $\alpha n \lg m + O(1)$ bits of space beyond the size of the WT and $\mathcal{A}$, for some constant $0 < \alpha \leq 1$.*

Proof. The procedure buildTemplate is used to create a strictly binary tree with m terminal null nodes ($\perp$), hence $m - 1$ actual nodes, of which $\lfloor m/2 \rfloor$ are leaves. Since the nodes are empty, just $O(m)$ time is needed.

For each character read, the algorithm visits $\lceil \lg m \rceil$ nodes on the path from the root to a leaf. At each node, a bit is appended to a growing bit vector, which can be done in constant amortised time by using dynamic arrays [5, Sec 17.4]. Therefore we have $O(n \lg m)$ time for filling the previously built template. Since $m = |\mathrm{supp}(T)| \leq |T| = n$, this phase dominates the cost, and we have $O(n \lg m)$ time for the entire construction process.

As for the space, notice that only the currently scanned symbol of T is used in the main loop, and so only that symbol needs to be kept in memory at any given time. Therefore we would only need space for the WT itself (bit arrays, pointers, etc.), plus the alphabet, plus a small constant amount of bits for the working variables. However, the dynamic arrays require extra space to ensure the constant amortised time per append operation. At any time, each level of the tree has as many *used* bits as the number of characters read so far, but at most α times as many bits may be physically allocated, with typical values of α ranging from 1.5 to 2. Hence at most $\alpha n \lg m + O(1)$ bits of total extra space may be required. ◀

▶ Remark. (i) In this analysis, we are not explicitly accounting for the work to build the auxiliary structures of the bit vectors, needed for constant-time rank/select. We can safely assume, however, that this work is linear on the size of the arrays, and thus the proposition remains valid. (ii) We also assume that the alphabet data type supports the computation of the index (rank) of a given character in constant time, which is easily accomplished by many dictionary data structures albeit with different space-time tradeoffs.

3.2 WT construction with unknown alphabet

We now turn to the online construction of the balanced WT of T with no prior knowledge about its support alphabet. Contrary to Algorithm 1, the WT cannot be laid out in advance

in this case. Instead, the alphabet, and consequently the shape of the tree itself, must be updated along with its content as the characters are scanned.

Let us consider first the update in the tree structure as the size m of the alphabet grows. The structure of the WT reflects an hierarchic binary decomposition of the alphabet, with an 1:1 correspondence between nodes and subalphabets. In order to keep the partition balanced as m increases, each *new* symbol will be successively assigned to either the left or right subalphabet in an alternate fashion, as with the previous case.

Now, consider the update from $W^{(i)} = W(T[0:i], \mathcal{A}^{(i)} = \text{supp}(T[0:i]))$ to $W^{(i+1)} = W(T[0:i+1], \mathcal{A}^{(i+1)} = \text{supp}(T[0:i+1]))$ upon reading t_i. If $t_i = a_j$ for some $a_j \in \mathcal{A}^{(i)}$, then the update is similar to one iteration of Algorithm 1, for the symbol is already represented in $W^{(i)}$. If, on the other hand, $t_i \notin \mathcal{A}^{(i)}$, then the update goes as follows. Let $s = |\mathcal{A}^{(i)}|$ be the size of the current alphabet. Starting at the root, if s is even (lsb=0), then the next symbol $a_s = t_i$ should be assigned to the left subalphabet. Thus the next position of the root bit array should be set to 0 and the left subtree should then be updated. If s is odd (lsb=1), we set $root.B[i]$ to 1 and proceed down to the right subtree. Appending a new bit to a bit vector does not affect its previous positions because of the alternating partitioning pattern. By following the same procedure at each node down the path, we may eventually reach an endpoint (a leaf in this case) corresponding to a binary subalphabet, say $\{a_p, a_q\}$ with $p < q < s$. After the addition of the bit concerning $t_i = a_s$ to this endpoint, the subalphabet becomes $\{a_p, a_q, a_s\}$, at which point this node has to be further split into a left child accounting for the binary subalphabet $\{a_p, a_s\}$, and an 'implicit' ($\perp$) right child corresponding to $\{a_q\}$. The new left child bit array should now represent $\text{proj}(T[:i+1], \{a_p, a_s\})$ with $a_p \equiv 0$ and $a_s \equiv 1$, thus being in the form 0^k1, for the only occurrence of a_s corresponds to the newly added t_i. Another possibility is that the endpoint represents a ternary alphabet $\{a_p, a_q, a_r\}$. In this case it would have a left child but not a right child, which is where the update should proceed to. So, a new right child has to be added to represent $\text{proj}(T[:i+1], \{a_q, a_s\})$. An endpoint cannot correspond to a subalphabet of size ≥ 3, or it would have split in previous iterations and the update could have continued to one of its children.

The procedure outlined above is given in Algorithm 2. The WT1(T) algorithm returns a reference to the root of the WT, as well as the support alphabet uncovered during the construction. It uses two main functions to incrementally build the WT. The update($root, \mathcal{A}, c$) function updates the bit vectors of the nodes in the appropriate root-to-leaf path of the current WT, adding information about t_i. As mentioned, this procedure is similar to one iteration of Algorithm 1. It returns a pointer to the endpoint *term* where the update stopped, plus the size *sterm* of the subalphabet represented by that node *after* the update. These values are fed into the testAndSplit procedure, which tests whether *term* needs to be further split and, if so, creates the appropriate child nodes.

▶ **Proposition 3.** *Algorithm 2 builds $W(T, \mathcal{A})$ in $O(n \lg m)$ time, using $\alpha n \lg m + O(1)$ bits of space beyond the size of the constructed WT and $\mathcal{A}$, for some constant $0 < \alpha \leq 1$.*

Proof. As in Algorithm 1, the total work amounts to creating the tree structure and filling the bit arrays. The only difference is that now these steps are interleaved rather than performed one after the other. The work required for creating the tree structure alone is the same in either case, since the resulting trees are isomorphic. As for the operations required for filling the bit arrays, we notice that every bit of the WT is set exactly once by a call to a bit array append operation, and is never modified thereafter. So, each bit is added in $O(1)$

Algorithm 2 Balanced WT online construction with unknown alphabet

```
1:  Algorithm WT1 (T = t_0 ··· t_{n−1})
2:      A ← {}
3:      root ← new WT node
4:      for i = 0, ..., n − 1 do
5:          term, sterm ← update(root, A, t_i)
6:          testAndSplit(term, sterm)
7:          A ← A ∪ {t_i}
8:      return root, A

1:  Algorithm testAndSplit(term, sterm)
2:      if sterm ≤ 2 then
3:          return
4:      chd ← new WT node
5:      b ← (sterm − 1)  mod 2
6:      k ← term.B.count(b) − 1
7:      chd.B.append(0^k)
8:      chd.B.append(1)
9:      if sterm = 3 then
10:         term.left ← chd
11:     else if sterm = 4 then
12:         term.right ← chd

1:  Algorithm update(root, A, c)
2:      if c ∈ A then
3:          s ← A.index(c)
4:          newc ← 0
5:      else
6:          s ← |A|
7:          newc ← 1
8:      cur ← root
9:      prev, sprev ← ⊥, 0
10:     while cur ≠ ⊥ do
11:         prev ← cur
12:         sprev ← s
13:         if s mod 2 = 0 then
14:             cur.B.append(0)
15:             cur ← cur.left
16:         else
17:             cur.B.append(1)
18:             cur ← cur.right
19:         s ← s ≫ 1
20:     return prev, sprev + newc
```

amortised cost by using dynamic arrays.[1] Hence we have the same $O(n \lg m)$ time for the entire construction procedure.

The space requirements analysis is identical to that of Proposition 2. ◀

4 Related work

As mentioned, the standard recursive procedure for constructing WTs requires $O(n \lg m)$ time. However the constants involved are somewhat large in practice, since it requires manipulating strings at each node. The space requirements are also significant because of the explicit projected copies of T at each level of the recursion, for a total $O(n \lg^2 m)$ bits in the worst case. As pointed out by some authors [19, 3], this space can be reduced to $O(n \lg m)$, on top of the original sequence, by reusing parts of the same allocated space through the recursion. This approach is implemented in LIBCDS [2]. The string manipulations remain costly nonetheless.

One way to reuse the same copy of the input sequence is by sorting their symbols according to the level of the recursion/WT. For instance, in a 'classic' WT, all the symbols whose highest bit is 0/1 will be to the left/right of the root. Thus, is we perform a stable, in-place sort of the sequence based on that bit, we will have all its symbols in the correct order for the next level, and the projections can now be represented by $[l : r]$ pairs. On the next level, the sort is based on the second highest bit, and so on. Tischler [25] explores these

[1] Actually, setting the initial bits of a newly created leaf (line 7 of procedure testAndSplit) may be a bit 'cheaper' in practice because we amortise the cost of getting to that node and, moreover, appending multiple zeros can be implemented a bit faster.

ideas to give space-efficient constructions of WTs in BFS and DFS order, using between constant and $O(\sqrt{n}(\lg m + \lg n))$ bits of extra space depending on a parameter c that also implies a $\lg c$ runtime multiplier. Claude and coauthors [3] also proposed space-efficient construction algorithms that are however more complex. Their most efficient algorithm runs in $O(n \lg n \lg^2 m + C(n \lg m))$ using $O(\lg m \lg n) + E(n \lg m))$ extra bits, where $C(r)$ and $E(r)$ denote the time to build and the space used by the auxiliary rank/select structures of the bit vectors. Such time and space are explicitly accounted for because the construction process depends on these structures. This algorithm is also destructive, meaning that it overwrites the original input sequence.

Simon Gog maintains SDSL, a very complete and mature C++ library of succinct data structures [9], containing the implementation of several methods described in the literature. This library has a few implementations of WTs with support to various topologies, alphabets, and bit vectors. The standard construction procedure consists in first reading the input to compute individual character counts and the effective support alphabet size, then initializing the tree structure, and finally filling in the node contents, much like in Algorithm 1.[2] Our novelty relative to this algorithm lies in the fact that our Algorithm 2 is online on the input and does not require precomputing the alphabet, which is both more general in theory, and can represent practical advantages, for example, in the context of streaming applications.

Some authors have recently proposed parallel algorithms for the construction of WTs. Fuentes-Sepúlveda and collaborators [7] explore the fact that the node corresponding to a certain character, at any given level of a classic WT, can be accessed by the highest bits of its index in the alphabet (as explained in Section 3.1), to build multiple levels in parallel. This $O(n \lg m)$ work is performed in $O(n)$ depth with $O(\lg m)$ processors, but the algorithm assumes continuous integers alphabet that need to be given as input. Shun [24] follows a similar ideas but builds the WT level-by-level, each level requiring $O(n)$ work in $O(\lg n)$ depth, for a total $O(\lg n \lg m)$ depth. These bounds are further improved by using stable sort algorithms to sort the characters of the input at each level of the tree by its highest bits so as to put them in the right order. However, this significantly increases memory demands.

Very recently, Munro and coauthors [13] presented the first algorithm to build a WT in $O(n\lceil \lg m/\sqrt{\lg n}\rceil)$, therefore a $\sqrt{\lg n}$-speedup over previous methods. Their technique is based on the use of bit-parallelism to pack the information concerning group symbols of the input sequence in words and process them together, thus achieving a time that is actually smaller than the size of the structure. Unfortunately, no implementation is available to the best of our knowledge [12].

5 Experimental analysis

In order to evaluate the applicability of our algorithm, we implemented a prototype in C and tested it with data obtained from the Pizza&Chili corpus website [20], as summarised in Table 1. For each data set, we created input files of sizes 2, 4, 8, 16, 32, 64, and 128 MB, by cropping the original files. These input sizes were shown to be enough for establishing a pattern in our experiments, as seen below. In addition, because the theoretical analysis of the previous section assumes the word-RAM model, these sizes also allow for a more clear comparison, unaffected by virtual secondary memory usage factors. We have then a total of $5 \times 7 = 35$ input sequences of varying length (n) and alphabet size (m), the two main parameters that affect the construction time and memory. We wanted to assess how our

[2] We have not found a description or analysis of this algorithm in the literature.

Table 1 Data sets used in the experiments. Original sources are indicated in [20].

Data set Id	Brief description	Total size	Alphabet size (m)	H_0 Entropy
dna	Gene DNA sequence	386 MB	16	5.465
proteins	Aminoacid sequence	1.2 GB	27	4.206
xml	XML bibliography data	283 MB	97	5.262
sources	C/Java source files	202 MB	230	5.537
english	English language texts	2.1 GB	239	4.525

implementation behaved in practice, relative to the theoretical predictions of Propositions 2 and 3.

Our implementation, herein identified as **fs**, consists in a simple pointer-based WT with $O(\lg m)$-time access, rank, and select operations, using a straightforward implementation of the uncompressed combined sampling rank and select bit vector [21]. We used no external libraries other than the C standard API. It is important to note that, although the alphabet is actually known for each data set in Table 1, this information is not fed into our algorithm. Instead, only the path to the text file is provided, which is read in one single pass as described in Section 3.2. In this case, the use of the C standard I/O (`stdio`) file manipulation functions [14] effectively results in a buffered input stream behaviour.

For comparison sake, we benchmarked our prototype together with other publicly available WT code. We wanted to assess the algorithms in a realistic situation where only the path to the file with the input is given. However, each implementation has different interfaces and so, for some of them, we wrote minimal wrapper code to read the input and pass it to the WT constructor using the appropriate internal data structures. We account for those operations as part of the algorithms to have a more uniform comparison. Here is a summary.

libcds. The LIBCDS [2] is a compressed data structures library whose WT code has been used in other comparative studies, including [3, 7]. The input has to be loaded into internal structures.

sdslil, sdslrrr. We used two variants of the WT construction example provided with the SDSL distribution [9]. The first variant, herein identified as sdslrrr, is identical to the example and uses RRR bit arrays[23], which is a compressed structure. Since our implementation does not use compression, we have also added a version based on uncompressed bit array, herein identified as sdslil. No input preprocessing was necessary.

pwt, dd. We used the code made available by the authors of [7]. The code requires that the text be encoded in a contiguous integer alphabet whose size has to be informed, and so we had to made the appropriate conversions.

swt. We used the code of the `serialWT` version made available by Shun [24], which gave the best results among the provided variants. No input conversion was needed.

When provided, the original scripts were used to build the executables. The only eventual modifications were made to ensure that each implementation was compiled with the same `-O3` optimisation flag. The actual code can be obtained from the address indicated at the end of this paper. All experiments reported here were performed on a portable computer equipped with a 2.0GHz Intel® Core™ i7-3537U CPU, 8 GB of RAM, and running 64-bit (Ubuntu 16.04 LTS) Linux ver. 4.4.0, with GCC ver. 5.4.0.

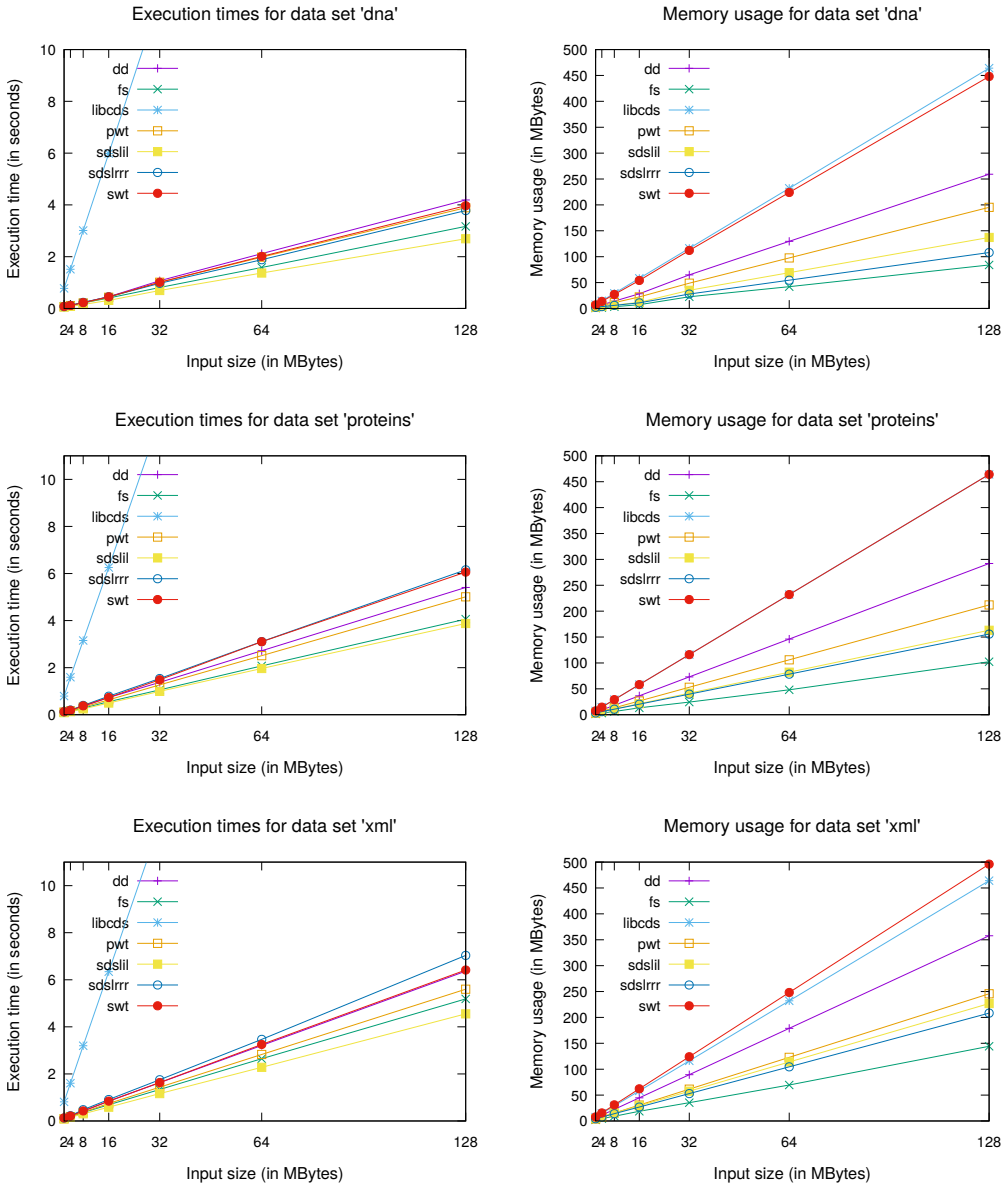

Figure 2 Experimental results (Part 1 – continued on next page). On the left, the average execution times, on the right, the corresponding memory usage information. Alphabet size (m) grows from top to bottom row.

5.1 Time experiments

We used the input files to build WT with all algorithms and measured the *total* execution time using the built-in Bash shell `time` command, including the user time, and the time spent by the system on behalf of the process. The tests were repeated 5 times for each input file and the average results are graphically summarised in the left column of Figure 2.

As it can be seen, a clear linear dependence on the length of the text is shown for all algorithms and for any given alphabet size. The logarithmic dependence on m is also fairly visible from the plots, by noticing, for instance, that de average times for the `dna` data

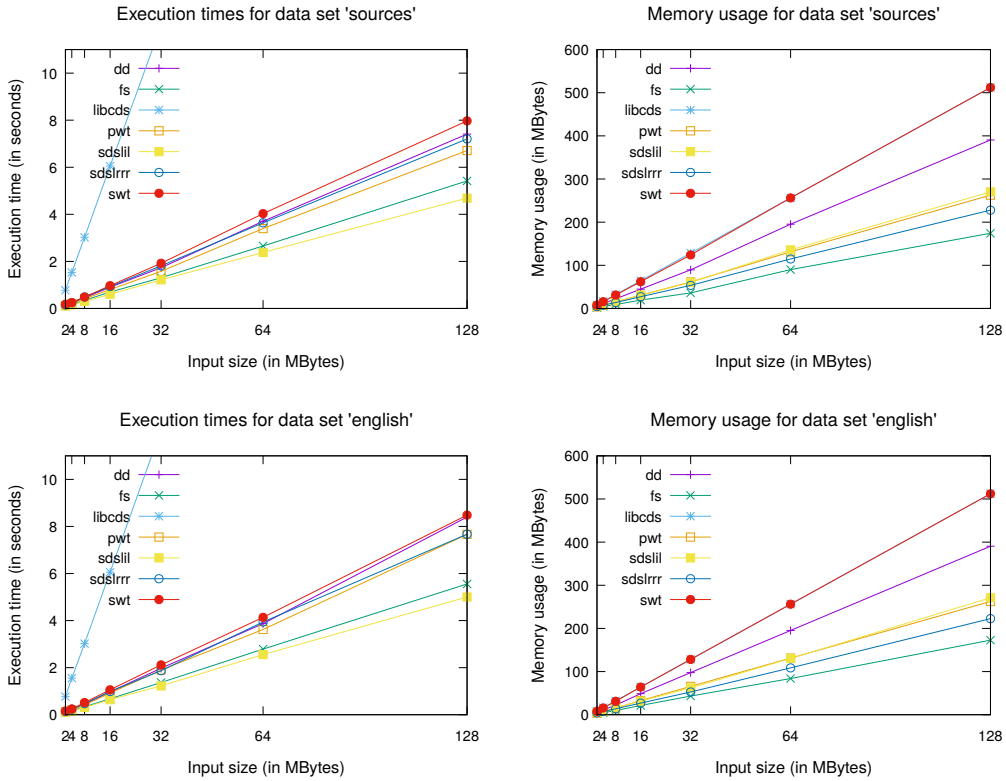

Figure 2 Experimental results (Part 2 – continued from previous page).

($\lg m = 4$) are roughly half of those for the english data set ($\lg m \approx 8$), with the notable exception of libcds. The results confirm the expected $O(n \lg m)$-time behaviour for (almost) all algorithms, although naturally not always with the same constant factors. Globally, all algorithms performed similarly (except libcds). sdslil was actually faster than the others, followed closely by our implementation.

In order to confirm the scalability of the method, we performed one last round of experiments with the largest input of Table 1, that is the 2.1GB english file. This time, we did not include the libcds method because it takes unreasonable time. As expected, the trend was maintained and sdslil was the fastest at 1m23s, followed by fs 1m31s. Other times were significantly higher: pwt 2m01s, sdslrrr 2m07s, dd 2m12s, and swt 2m39s.

5.2 Memory experiments

We also measured the memory usage of all algorithms on each input file using the `massif` tool of the `valgrind` package [22]. Since all algorithms are deterministic, the memory consumption does not vary between executions with the same input. The results are summarised in the right column of Figure 2. Shown are the peak total heap usage in MB during each execution.

As with time, there is a clear linear dependence on the text length for all algorithms, for any given alphabet size. The logarithmic dependence on the alphabet size is still clearly visible for some, in particular fs, but not all implementations. It is important to observe that the results shown in Figure 2 comprise the space taken by the WT itself, and the extra working space used by the construction algorithm. The fact that some implementations, like

Table 2 Average memory overheads (standard deviation in parenthesis).

Data set	Algorithm			
	dd	fs	libcds	pwt
dna	2.76 (0.25)	0.34 (0.04)	6.27 (0.02)	1.90 (0.13)
proteins	2.84 (0.00)	0.33 (0.04)	5.12 (0.02)	1.79 (0.00)
xml	2.39 (0.00)	0.37 (0.03)	3.41 (0.01)	1.33 (0.00)
sources	1.93 (0.12)	0.24 (0.11)	3.09 (0.01)	1.00 (0.06)
english	1.98 (0.13)	0.29 (0.07)	3.06 (0.01)	1.02 (0.07)
	sdslil	sdslrrr	swt	
dna	0.99 (0.32)	0.72 (0.28)	5.88 (0.11)	
proteins	1.39 (0.30)	1.28 (0.30)	5.12 (0.02)	
xml	1.33 (0.22)	1.15 (0.22)	3.71 (0.02)	
sources	1.10 (0.16)	0.85 (0.16)	3.00 (0.05)	
english	1.11 (0.14)	0.80 (0.15)	3.01 (0.05)	

libcds and swt, use much more space than the others, while representing the same information, and, moreover, that this working memory varies very little with the alphabet size, suggest that the total space is dominated by structures used by the construction algorithm which depend essentially on n.

We also estimated the relative *space overhead*, defined as the ratio $(M - S)/S$, where $S = m \lg n$ bits is a lower bound of the space taken by the uncompressed WT, and M is the total memory measured in the experiments. The average overheads per algorithm and per data set are shown in Table 2. The numbers confirm that the extra memory required by fs was comfortably under the predicted upper bound of 50% due to its use of a $\alpha = 1.5$ growth factor for the dynamic bit arrays. In all tests, our implementation outperformed the others in that respect, seconded only by sdslrrr, which uses compression, with 2–3 times more overhead.

Finally, we also performed one last test to gauge memory consumption using the 2.1GB english file. Our method performed better than all the others by similar margins. The fs implementation used 2.80GB, seconded by sdslrrr, which used 3.57GB, followed by pwt 4.22GB, sdslil 4.34GB, dd 6.27GB, and swt 8.23GB.

6 Discussion

We have presented a method for the online construction of the balanced wavelet tree of a source text T requiring $O(n \lg m)$ time and very little working memory. No previous information about the alphabet is assumed. We argue that our algorithm is conceptually simpler than most other methods but, despite its simplicity, it compares quite well in practice against other implementations, as shown by a series of experiments on real data, offering an appealing time vs. space compromise, not to mention that the online characteristic makes it more amenable to certain applications.

We regret not having an implementation of [13] to compare, but we note that, even for inputs of one petabyte (2^{50} bytes), the theoretical speedup would be of just about 7×. This margin can be easily eroded in practice by a more complex code. We equally regret not having found the space-efficient implementation `PERMUTE` [3] although the asymptotic costs and the reported comparisons of this algorithm against LIBCDS suggest that our implementation could still be a more favourable compromise.

Our current implementation is pointer-based, requiring $O(m \lg n)$ bits of space for the tree topology alone, which can be a drawback for applications with large alphabets. To mitigate this problem, we notice that Algorithm 2 builds the WT one level at a time, and so we can represent its structure implicitly, at any time, by using an array $P[0 : h]$ of $h = 2^{\lceil \lg s \rceil} - 1$ positions, where s stands for the number of characters currently represented. As in the binary heap [5, Sec 6.1], we assign the root to position 0 and, from there on, the left and right children of the node at position i are associated to positions $2i + 1$ and $2i + 2$, respectively. We let $P[i]$ store the (pointer to the) bit vector of the node corresponding to position i. In this case, P, and hence the WT, is actually a dynamic array of bit vectors which is doubled every time a node is first added to a new level. This way we still have constant amortised time per node creation and we no longer need the node pointers. However we may yet have up to $s - 2$ unused positions of P, corresponding to the incomplete lowest level of the WT. So, the overall space is about the same in the worst case as with node pointers, but it may be more space-efficient in practice since we expect the lowest level to be fairly populated, and the unused positions at the end of P can be easily trimmed off after the construction.

Notice also that, in the test data sets of Table 1, the alphabet size was always under 256, so that it could be efficiently represented as a simple byte array, whereas in applications with large alphabets, the representation of the alphabet itself can be an issue. However, this is arguably a separate problem not particular to our construction method. In fact Algorithm 2 is reasonably oblivious to the actual data structure used, other than by supposing that it is a form of dictionary that supports insertion and membership queries in constant time. Notice that the space required by the alphabet was consciously excluded from the space complexity of Proposition 3.

Another limitation of our method is that it imposes an order on the alphabet symbols by somehow sorting them according to their order of appearance on the represented string. By contrast, the traditional balanced WT construction recursively partitions the alphabet in halves, respecting a certain 'natural' order, e.g. the lexicographic order for character alphabets, or the ascending order for integer alphabets. While our approach is coherent with the interpretation that the alphabet is 'unknown' (and hence so is its natural order), and this does not present a problem for the rank, select, or access operations, it may void the use of the WT on applications that assume a specific order for the alphabet, like in range quantile queries [8].

The first algorithm presented in this paper can be adapted to other WT topologies like the Huffman WT [18]. If we know the alphabet and relative character frequencies, then we can build the template tree following the greedy $O(m \lg m)$ Huffman tree construction algorithm [5, Sec 16.3] and then fill its contents just like in Algorithm 1. Even if the alphabet is unknown, we can still first build the template with one pass over T and then fill it in a second pass. It remains to be shown whether the single-pass online method could be adapted to the Huffman WT within the same time bounds.

Finally, we remark that the online construction procedure shown in Algorithm 2 suggests the static case where the input string/stream is read to its end to fill the raw bit contents of the nodes, before the WT is ever used. These raw bit vectors are then usually processed in linear time to produce a sub-linear amount of supporting information that allow for constant-time binary rank and select queries. However, we notice that the construction method is independent of the actual bit vector implementation. All it supposes is that the bit vectors support constant (amortised) time append operations so that Proposition 3 holds. In particular, if dynamic rank/select bit vectors are used [16, 4] then this construction method yields a partially dynamic WT implementation that allows for rank, select, and access operations to be performed at any time on a prefix of the input sequence.

7 Availability

The source code, as well as the experimental data and scripts used in this paper can be obtained from `http://www.cin.ufpe.br/~paguso/sea2017`.

Acknowledgments. We thank the anonymous reviewers for their thoughtful comments.

References

1. David Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.
2. Francisco Claude. LIBCDS: A compressed data structures library (https://github.com/fclaude/libcds2). URL: `https://github.com/fclaude/libcds2`.
3. Francisco Claude, Patrick K. Nicholson, and Diego Seco. Space Efficient Wavelet Tree Construction. In *Proceedings of the 18th Symposium on String Processing and Information Retrieval - SPIRE 2011*, pages 185–196, Pisa, 2011. `doi:10.1007/978-3-642-24583-1_19`.
4. Joshimar Cordova and Gonzalo Navarro. Practical Dynamic Entropy-Compressed Bitvectors with Applications. In *Proceedings of the 15th International Symposium on Experimental Algorithms – SEA 2016*, pages 105–117, 2016. `doi:10.1007/978-3-319-38851-9_8`.
5. Thomas H. Cormen, Charles E. Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction To Algorithms*. MIT Press, 1990.
6. Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. The myriad virtues of Wavelet Trees. *Information and Computation*, 207(8):849–866, 2009. `doi:10.1016/j.ic.2008.12.010`.
7. José Fuentes-Sepúlveda, Erick Elejalde, Leo Ferres, and Diego Seco. Efficient Wavelet Tree Construction and Querying for Multicore Architectures. In *Proceedings of the 13th International Symposium on Experimental Algorithms – SEA 2014*, pages 150–161, Copenhagen, 2014. `doi:10.1007/978-3-319-07959-2_13`.
8. Travis Gagie, Simon J. Puglisi, and Andrew Turpin. Range Quantile Queries: Another Virtue of Wavelet Trees. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval – SPIRE 2009*, pages 1–6, Saariselkä, 2009. `arXiv:arXiv:0903.4726v6`, `doi:10.1007/978-3-642-03784-9_1`.
9. Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From Theory to Practice: Plug and Play with Succinct Data Structures. In *Proceedings of the 13th International Symposium on Experimental Algorithms – SEA 2014*, pages 326–337, Copenhagen, 2014. `doi:10.1007/978-3-319-07959-2_28`.
10. Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-Order Entropy-Compressed Text Indexes. In *Proceedings of the 14th annual ACM-SIAM Symposium On Discrete Algorithms – SODA 2003*, pages 841–850, Philadelphia, 2003.
11. J. Ian Munro. Tables. In *Proceedings of the International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 37–42, Hyderabad, 1996. `doi:10.1007/3-540-62034-6_35`.
12. J. Ian Munro. Personal communication, 2017.
13. J. Ian Munro, Yakov Nekrich, and Jeffrey S. Vitter. Fast construction of wavelet trees. *Theoretical Computer Science*, 638:91–97, 2016. `doi:10.1016/j.tcs.2015.11.011`.
14. ISO/IEC. Information technology – Programming languages – C. *ISO/IEC 9899:2011 Std*, 2011.
15. Guy Jacobson. *Succinct static data structures*. PhD thesis, Carnegie Mellon University, 1989.

16 Veli Mäkinen and Gonzalo Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms*, 4(3):1–38, 2008. doi:10.1145/1367064.1367072.

17 Christos Makris. Wavelet trees: A survey. *Computer Science and Information Systems*, 9(2):585–625, 2012. doi:10.2298/CSIS110606004M.

18 Gonzalo Navarro. Wavelet Trees for All. In *Proceedings of the 23rd Annual conference on Combinatorial Pattern Matching – CPM 2012*, pages 2–26, Helsinki, 2012. doi:10.1007/978-3-642-31265-6_2.

19 Gonzalo Navarro. *Compact data structures: a pratical approach*. Cambridge Univ Press, 2016.

20 Gonzalo Navarro and Paolo Ferragina. Pizza&Chili Corpus Website (http://pizzachili.dcc.uchile.cl/). URL: http://pizzachili.dcc.uchile.cl/.

21 Gonzalo Navarro and Eliana Providel. Fast, Small, Simple Rank/Select on Bitmaps. In *Proceedings of the 11th international Symposium on Experimental Algorithms – SEA 2012*, pages 295–306, Bordeaux, 2012. doi:10.1007/978-3-642-30850-5_26.

22 Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation – PLDI 2007*, pages 89–100, New York, 2007. doi:10.1145/1250734.1250746.

23 Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees, Prefix Sums and Multisets. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms – SODA 2002*, pages 233–242, San Francisco, 2002. arXiv:arXiv:0705.0552v1.

24 Julian Shun. Parallel Wavelet Tree Construction. In *2015 Data Compression Conference*, pages 63–72. IEEE, apr 2015. doi:10.1109/DCC.2015.7.

25 German Tischler. On wavelet tree construction. In *Proceedings of the 22nd annual conference on Combinatorial pattern matching – CPM 2011*, pages 208–218, Palermo, 2011.