Trickier XBWT Tricks

Enno Ohlebusch(✉), Stefan Stauß, and Uwe Baier

Institute of Theoretical Computer Science, Ulm University, 89069 Ulm, Germany
{Enno.Ohlebusch,Stefan.Stauss,Uwe.Baier}@uni-ulm.de

Abstract. A trie [11] is one of the best data structures for implementing and searching a dictionary. However, to build the trie structure for larger collections of strings takes up a lot of memory. Since the eXtended Burrows-Wheeler Transform (XBWT) [8,9] is able to compactly represent a labeled tree, it can naturally be used to succinctly represent a trie. The XBWT also supports navigational operations on the trie, but it does not support failure links. For example, the Aho-Corasick algorithm [1] for simultaneously searching for several patterns in a text achieves its good worst-case time complexity only with the aid of failure links. Manzini [18] showed that a balanced parentheses sequence P can be used to support failure links in constant time with only $2n + o(n)$ bits of space, where n is the number of internal nodes in the trie. Besides practical algorithms that construct the XBWT, he also provided two different algorithms that construct P. In this paper, we suggest an alternative way for constructing P that outperforms the previous algorithms.

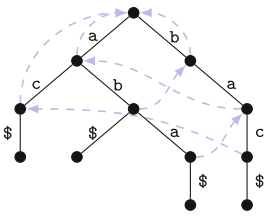
1 Introduction

The eXtended Burrows-Wheeler Transform (XBWT) [8,9] can be used to compactly represent a trie by a character array L and a bit array Last; see Fig. 1 for an example. A recent empirical comparison [19] of string dictionary implementations shows that the XBWT achieves the best compression of all techniques under consideration. Moreover, in contrast to most other methods, the XBWT supports substring searches. The compact representation of the XBWT can be computed as follows. Each internal node v of the trie T is associated with a string that is obtained by concatenating the characters at the edges in the *upward* path from v to the root of T (the root itself is associated with the empty string ε). If T has n internal nodes, then there are n associated strings and the (virtual) array $\Pi[1..n]$ stores them in lexicographical order. We (conceptually) number the internal nodes of T according to Π: If node v is associated with the string $\Pi[i]$, it gets the number i. Let L_i be the set of characters at outgoing edges of node i (in no particular order) and let the character array L contain the concatenation of $L_1, L_2, \ldots, L_n$. Furthermore, the bit array Last stores the borders of L_i: we initialize Last with zeros and for all $i \in \{1, \ldots, n\}$ we set $\mathsf{Last}[j_i] = 1$, where $j_i = \sum_{\ell=1}^{i} |L_\ell|$. As already mentioned, the XBWT representation of the trie consists of the arrays L and Last. These arrays can be calculated with the help of an array MR, which we will define next.

T. Gagie et al. (Eds.): SPIRE 2018, LNCS 11147, pp. 325–333, 2018.
https://doi.org/10.1007/978-3-030-00479-8_26

Suppose the trie T is constructed from the pairwise distinct strings $x_1, \ldots, x_k$. Let $y_i = x_i^R$, where x_i^R denotes the string that is obtained by reversing x_i. Furthermore, let $S = y_1\$y_2\$\ldots y_k\$$ be the concatenation of the y_i, separated by a special character \$, which is assumed to be smaller than any other character. In the following, let m be the length of S (note that $m = k + \sum_{i=1}^{k} |x_i|$). The suffix array SA and the Burrows-Wheeler Transform BWT of S are obtained by sorting the suffixes of S lexicographically (this can be done in linear time): If $S[j..m]$ is the i-th lexicographically smallest suffix of S, then $\mathsf{SA}[i] = j$ and $\mathsf{BWT}[i] = S[j-1]$ is the character preceding that suffix (if $j = 1$, then $\mathsf{BWT}[i] =$ \$); see Fig. 1 for an example. Fast implementations of (semi-) external suffix sorting algorithms exist [6,16], but multi-string BWT construction algorithms may be competitive in the context of this paper; see [3,17]. The array MR is defined with the help of suffix array intervals. If ω is a substring of S, then the ω-interval is the largest interval $[i..j]$ such that ω is a prefix of all the suffixes in the interval $[i..j]$. Now $\mathsf{MR}[lb] = 1$ if and only if lb is the left boundary of a z-interval, where z is a suffix of some y_i (i.e., z^R is a prefix of some x_i). Note that $\mathsf{MR}[1] = 1$ because ε is a suffix of all y_i and $[1..m]$ is the ε-interval. To avoid case distinctions, we set $\mathsf{MR}[m+1] = 1$. Let $j_1 = 1 < j_2 < \cdots < j_n < j_{n+1} = m+1$ be the indices with $\mathsf{MR}[j_\ell] = 1$. For each i with $1 \le i \le n$, the interval $[j_i..j_{i+1}-1]$ is the $\Pi[i]$-interval. Thus L_i is the set of the characters in $\mathsf{BWT}[j_i..j_{i+1}-1]$ and $\mathsf{Last}[p_i] = 1$, where $p_i = \sum_{\ell=1}^{i} |L_\ell|$. It is readily verified that the arrays L and Last can be computed in $O(m)$ time by simultaneously scanning the arrays MR and BWT from left to right.

Last	L	Π
0	a	ϵ
1	b	
0	b	a
1	c	
1	c	ab
1	\$	aba
1	a	b
0	a	ba
1	\$	
1	\$	ca
1	\$	cab



i	MR	C_c	BWT	sorted suffixes
1	1	0	a	\$
2	0	0	b	\$aba\$
3	0	0	a	\$ca\$cab\$aba\$
4	0	0	a	\$cab\$aba\$
5	1	0	b	a\$
6	0	0	b	a\$ca\$cab\$aba\$
7	0	0	c	a\$cab\$aba\$
8	1	0	c	ab\$aba\$
9	1	3	\$	aba\$
10	1	0	a	b\$aba\$
11	1	0	a	ba\$
12	0	2	\$	ba\$ca\$cab\$aba\$
13	1	0	\$	ca\$cab\$aba\$
14	1	3	\$	cab\$aba\$

Fig. 1. Example for the input strings ab, ac, bac, aba. Left: XBWT consisting of the arrays Last and L (the array Π is not stored). Center: Trie of the strings, where failure links of internal nodes are indicated by dashed arrows. Right: MR, C_c, and BWT for the concatenation of the reversed strings (i.e., $S =$ ba\$ca\$cab\$aba\$).

Recall that node i in the trie T is associated with the string $\Pi[i]$. In this context, the *failure link* of i points to the node j so that $\Pi[j]$ is the longest proper *prefix* of $\Pi[i]$. Failure links are not supported by the XBWT representation of T, but Manzini [18] showed that a balanced parentheses sequence P can be used to

support them in constant time with only $2n+o(n)$ bits of space. P can be defined by means of Π: For $i = 1, \ldots, n$ a pair of parentheses is written by repeating the following: (1) For each $\ell < i$, for which its closing parenthesis has not been written yet and $\Pi[\ell]$ is *not* a prefix of $\Pi[i]$, write a closing parenthesis. (2) Write the opening parenthesis for i. After termination of this for-loop, write a closing parenthesis for each ℓ, for which its closing parenthesis has not been written yet. In the example of Fig. 1, we have $P = ((((()))(()))(()))$. P can be preprocessed in linear time, using only $o(n)$ bits, so that the operations *rank*, *select*, and *enclose* can be supported in constant time [7,15,20]. Using these operations on P, failure links can be supported in constant time; see [18, Lemma 4] for details. Manzini [18] devised two different algorithms that construct P. In the next section, we suggest an alternative way for constructing P that outperforms his algorithms.

2 The New Algorithm

Our new construction algorithm uses an idea of Belazzougui [4], who devised a rather simple method to build the balanced parenthesis representation of a suffix tree topology. He writes: "Our key observation is that we can easily build a balanced parenthesis representation by enumerating the suffix array intervals. More precisely for every position in $[1..n]$, we associate two counters, one for open and the other for close parentheses implemented through two arrays of counters $C_o[1..n]$ and $C_c[1..n]$. Then given a suffix array interval $[i, j]$ we will simply increment the counters $C_o[i]$ and $C_c[j]$. Then we scan the counters C_c and C_o in parallel and for each i from 1 to n, write $C_o[i]$ opening parentheses followed by $C_c[i]$ closing parentheses. It is easy to see that the constructed sequence is that of the balanced parentheses of the suffix tree." Since we do not want to represent a suffix tree topology, we cannot enumerate *all* suffix array intervals. Instead, we must enumerate all z-intervals for which z is a suffix of some y_i (for then z^R is a prefix of some x_i). Recall that $\mathsf{MR}[lb] = 1$ if and only if lb is the left boundary of such a z-interval. Consequently, the array $\mathsf{MR}[1..m]$ coincides with the array $C_o[1..m]$. Moreover, observe that if z is a suffix of some y_i, then the left boundary b_z of the z-interval in the suffix array of S coincides with the left boundary $b_{z\$}$ of the $z\$$-interval because $z\$$ is a substring of S and $\$$ is the smallest character.

For the explanation of the pseudo-code of our new construction algorithm (Algorithm 1), we need a few preliminaries. For each character c, $C[c]$ is the overall number of occurrences of characters in $\mathsf{BWT}[1..m]$ that are strictly smaller than c. Given the ω-interval $[lb..rb]$ and a character c, the $c\omega$-interval $[i..j]$ can be computed by $i = C[c] + rank_c(\mathsf{BWT}, lb-1) + 1$ and $j = C[c] + rank_c(\mathsf{BWT}, rb)$, where $rank_c(\mathsf{BWT}, lb-1)$ returns the number of occurrences of character c in the prefix $\mathsf{BWT}[1..lb-1]$ (we have $i \leq j$ if $c\omega$ is a substring of S; otherwise $i > j$); see [10] for details. The (balanced) *wavelet tree* [14] of the BWT supports such a backward search step in $O(\log \sigma)$ time, where σ is the size of the alphabet. Backward search can be generalized on the wavelet tree as follows: Given an ω-interval $[lb..rb]$, a slight modification of the procedure $getIntervals([lb..rb])$

Algorithm 1. Computation of the arrays MR and C_c

```
1: function VISIT(b_{z$}, e_{z$}, e_z)      ▷ the z$-interval is [b_{z$}..e_{z$}] and the z-interval is
   [b_{z$}..e_z], where z is a suffix of some string y_i
2:     MR[b_{z$}] ← 1
3:     C_c[e_z] ← C_c[e_z] + 1
4:     list ← getIntervals([b_{z$}..e_{z$}])
5:     for each (c, [b_{cz$}..e_{cz$}]) with c ≠ $ in list do
6:         if e_z = e_{z$} then
7:             e_{cz} ← e_{cz$}
8:         else
9:             e_{cz} ← C[c] + rank_c(BWT, e_z)
10:        VISIT(b_{cz$}, e_{cz$}, e_{cz})
```

described in [5] returns the list $[(c, [i..j]) \mid c\omega$ is a substring of S and $[i..j]$ is the $c\omega$-interval], where the first component of an element $(c, [i..j])$ is a character. The worst-case time complexity of the procedure *getIntervals* is $O(occ + occ \cdot \log(\sigma/occ))$, where occ is the number of elements in the output list; see [12, Lemma 3].

If $z = \varepsilon$ is the empty string, then the z-interval is $[b_z..e_z] = [1..m]$ and the z\$-interval is $[b_{z\$}..e_{z\$}] = [1..k]$. The function call VISIT $(1, k, m)$ computes the arrays $\mathsf{MR} = C_o$ and C_c; the pseudo-code of this function can be found in Algorithm 1. The function first counts an opening parenthesis at position $b_z = b_{z\$}$ and a closing parenthesis at position e_z. With the help of the procedure *getIntervals* it then computes all non-empty cz\$-intervals, where $c \in \Sigma$ and $c \neq \$$. The fact that a cz\$-interval $[b_{cz\$}..e_{cz\$}]$ is not empty means that cz is a suffix of some y_i. It follows as a consequence that the cz-interval $[b_{cz}..e_{cz}]$ is also not empty. Again, $b_{cz} = b_{cz\$}$ holds true, but the right boundary e_{cz} of the cz-interval is not known yet. Now there are two cases. If the right boundaries of the z-interval and the z\$-interval coincided, then so do the right boundaries e_{cz} and $e_{cz\$}$ of the cz-interval and the cz\$-interval. If they were not the same, e_{cz} must be computed by evaluating $C[c] + rank_c(\mathsf{BWT}, e_z)$ as in backward search. Finally, the function recursively calls itself with the new parameters $b_{cz\$}, e_{cz\$}, e_{cz}$.

The overall time complexity of the construction of P is $O(m \log \sigma)$ because the BWT can be build in $O(m)$ time, the wavelet tree of the BWT can be constructed in $O(m \log \sigma)$ time, initialization and computation of the arrays MR and C_c takes $O(m + n \log \sigma)$ time (n is the number of internal nodes of the trie and satisfies $n \leq m$), and the computation of P based on MR and C_c requires $O(m)$ time.

Let us consider the working space of Algorithm 1. By the definition of P, there are at most $\max_i |x_i|$ consecutive closing parentheses, thus the array C_c requires $m \log(\max_i |x_i|)$ bits. The array MR occupies only m bits and the wavelet tree of the BWT essentially uses $m\lceil \log \sigma \rceil + o(m \log \sigma)$ bits of space; see e.g. [21]. The stack for the recursion contains (at any point in time) at most $\max_i |x_i|$ elements. Each stack element stores a list returned by the procedure *getIntervals*; this

Algorithm 2. Computation of P with less space

```
1: input: BWT
2: compute the bit array MR of size m
3: preprocess MR so that rank-queries can be answered in constant time
4: initialize an array C'_c of size n = rank_1(MR, m) with zeros
5: VISIT2(1, k, m)
6: initialize an array P of size 2n with zeros                ▷ 2n opening parentheses
7: k ← 1
8: for i ← 1 to n do
9:     k ← k + 1                          ▷ opening parenthesis because P[k] = 0
10:    for j ← 1 to C'_c[i] do
11:        P[k] ← 1                                  ▷ write closing parentheses
12:        k ← k + 1
13: return: P
```

list contains at most σ elements of the form $(c, [lb..rb])$. Since every list element requires $O(1)$ space, the whole stack uses $O(\sigma \cdot \max_i |x_i|)$ space.

3 Saving Space

As already observed by Manzini [18], the number n of ones in the bit array MR gives the number of internal nodes of the trie. If one computes MR in a first phase (for instance, by the algorithm in [18, Fig. 4]), then n is known and more space-efficient algorithms for computing P can be deduced. Manzini suggests to use two arrays RCP' and LEN' of length n that use $O(n \log(\max_i |x_i|))$ bits of memory and store only values for which the corresponding entry in the array MR equals 1. His algorithm [18, Fig. 5] calculates P based on these arrays. We would like to follow this approach, but the example from Fig. 1 shows that there are non-zero entries $C_c[i]$ for which $\mathsf{MR}[i] = 0$ ($i = 12$ in Fig. 1). We next derive a version of Algorithm 1 that increments only counters at indices i for which $\mathsf{MR}[i] = 1$. To distinguish the new version from Algorithm 1, we use $\hat{C}_c$ to denote the array of counters (which is still of size m). Recall that Algorithm 1 increments $C_c[e_z]$ by one, where e_z is the right boundary of a z-interval. The new version increments $\hat{C}_c[j]$ instead, where $j = \max\{i \mid i \leq e_z \text{ and } \mathsf{MR}[i] = 1\}$. In other words, if $\mathsf{MR}[e_z] = 1$, it increments $\hat{C}_c[e_z]$ and if $\mathsf{MR}[e_z] = 0$, it increments the counter at which the previous one in MR can be found. In the example from Fig. 1 it would increment the counter at index 11 instead of that at $i = 12$. To see that this preserves correctness, consider two indices i and j so that $\mathsf{MR}[i] = 1$, $\mathsf{MR}[j] = 1$, and $\mathsf{MR}[k] = 0$ for all k with $i < k < j$ (the case in which i is the last index with $\mathsf{MR}[i] = 1$ follows similarly). On the one hand, if we use the array C_c, an opening parenthesis will be written for $\mathsf{MR}[i] = 1$, followed by $\sum_{k=i}^{j-1} C_c[k]$ closing parentheses, and then an opening parenthesis will be written for $\mathsf{MR}[j] = 1$. On the other hand, if we use the array $\hat{C}_c$, an opening parenthesis will be written for $\mathsf{MR}[i] = 1$, followed by $\hat{C}_c[i]$ closing parentheses and an opening parenthesis for $\mathsf{MR}[j] = 1$. Since $\hat{C}_c[i] = \sum_{k=i}^{j-1} C_c[k]$, it follows that both algorithms compute

the same sequence of parentheses. Algorithm 2 implements the new version of Algorithm 1, however, it uses an array C'_c of length n and size $n \log(\max_i |x_i|)$ bits instead of the array $\hat{C}_c$ of length m. First, it computes the bit array MR and then preprocesses it so that *rank*-queries can be answered in constant time. Then it calls the function VISIT2 with parameters $1, k, m$. Function VISIT2 can be obtained from function VISIT by deleting line 2 in Algorithm 1 and replacing the assignment in line 3 by $C'_c[rank_1(\mathsf{MR}, e_z)] \leftarrow C'_c[rank_1(\mathsf{MR}, e_z)] + 1$. That is, for a z-interval $[b_z..e_z]$, function VISIT2 increments $C'_c[rank_1(\mathsf{MR}, e_z)]$ by one. This simulates the new version of Algorithm 1, in which $\hat{C}_c[j]$ is incremented, where $j = \max\{i \mid i \leq e_z \text{ and } \mathsf{MR}[i] = 1\}$.

In contrast to Algorithm 1, Algorithm 2 uses two passes to compute MR and C'_c separately. That is, it saves space by using C'_c instead of C_c, but the run-time doubles in practice (its time complexity is also $O(m \log \sigma)$).

4 Experimental Results

We experimentally compared our new XBWT construction algorithms with the ones presented in [18]. More precisely, we implemented the following algorithms, as we could not find an implementation of Manzini's algorithms:

- `MAN` : algorithm by Manzini [18, Sect. 4]
- `MAN-LW` : lightweight algorithm by Manzini [18, Sect. 4]
- `OSB` : our new algorithm (Sect. 2)
- `OSB-LW` : lightweight version of the new algorithm (Sect. 3)

Our test data—the files dblp.xml, dna, proteins, english, and sources—originate from the Pizza & Chili corpus.[1] In our experiments, we constructed tries for each of the files using the above-mentioned algorithms, where the distinct lines of a file were used as input strings for trie construction.

Table 1. Trie construction results. The left column lists test data along with its size and the length of its longest string. The other columns show, for each test case, the construction time in seconds and the memory peak during construction, excluding suffix array and BWT construction.

File	MAN		MAN-LW		OSB		OSB-LW	
	time	peak	time	peak	time	peak	time	peak
dblp.xml 165 MB $\max \lvert x_i \rvert = 685$	66 s	784 MB	122 s	392 MB	39 s	430 MB	68 s	310 MB
dna 384 MB $\max \lvert x_i \rvert = 28{,}515{,}262$	242 s	1,873 MB	885 s	2,741 MB	252 s	1,530 MB	510 s	1,542 MB
proteins 864 MB $\max \lvert x_i \rvert = 36{,}805$	1,247 s	4,322 MB	1,942 s	3,896 MB	572 s	2,594 MB	1,102 s	2,394 MB
english 1,485 MB $\max \lvert x_i \rvert = 2{,}792$	961 s	7,613 MB	5,256 s	5,858 MB	1,600 s	4,592 MB	3,155 s	4,135 MB
sources 143 MB $\max \lvert x_i \rvert = 1{,}491$	65 s	680 MB	189 s	448 MB	53 s	396 MB	99 s	326 MB

[1] http://pizzachili.dcc.uchile.cl.

Algorithm 3. Computation of P by a depth-first traversal of a generalized ST

```
1: function DFT(v)
2:     if v is leaf and its incoming edge has a label ≠ $ then
3:         write an opening parenthesis and a closing parenthesis
4:     if v is an internal node then
5:         if v has an outgoing edge with label $ then write an opening parenthesis
6:         for each child node w of v (in lexicographical order of the labels of the
   outgoing edges from v) call DFT(w)
7:         if v has an outgoing edge with label $ then write a closing parenthesis
```

The experiments were conducted on a 64 bit Ubuntu 16.04.4 LTS system equipped with two 16-core Intel Xeon E5-2698v3 processors and 256 GB of RAM. All programs were compiled with the O3 option using g++ (version 5.4.1). Our programs and the benchmark are publically available.[2] Table 1 shows the results of the experiments. Among all tested algorithms, `OSB-LW` has the lowest memory peak. Surprisingly, if the trie is built from long strings (`dna`), Algorithm `MAN-LW` requires a lot of memory, probably because of a stack that stores items consisting of several components. Algorithms `MAN` and `OSB` are the fastest construction methods, but despite of a lower memory peak, `OSB` often outperforms `MAN` (we think this is caused by cache-misses in `MAN`, which occur during accesses to the suffix array and the test data).

Summing up, our new algorithms `OSB` and `OSB-LW` outperform the algorithms `MAN` and `MAN-LW` in terms of memory consumption, and perform similarly fast or even faster. As `OSB` requires only a little more memory than `OSB-LW`, but performs similarly fast as `MAN`, algorithm `OSB` has a good space-time tradeoff and therefore is our method of choice for XBWT construction.

Our implementation is based on the `sdsl-lite` library [13] and we further tried to reduce the memory peak of our algorithms by using compressed wavelet trees supported by the `sdsl-lite` library. With Huffman-shaped wavelet trees that use rrr-bitvectors [13], it is possible to obtain a 25% reduction of the memory peak on average, but the construction time increases by a factor of 2.5 on average. It might be worth trying other compressed wavelet trees such as the one described in [2], but unfortunately its implementation contained in the `sdsl-lite` library lacks support for the procedure $getIntervals$.

5 Concluding Remark

Some readers may prefer to construct the balanced parentheses sequence P by means of a suffix tree, and of course this is possible. To this end, build the generalized suffix tree ST of the reversed input strings $y_1, y_2 \ldots, y_k$. In such a generalized suffix tree, all strings are either terminated by \$ or they are terminated by pairwise different symbols $\$_1, \$_2 \ldots, \$_k$. Here, we will use \$. Then traverse ST in a depth-first fashion, i.e., call function DFT of Algorithm 3 with

[2] https://www.uni-ulm.de/in/theo/research/seqana/.

the root of ST as parameter. If an internal node v is visited during the traversal, the algorithm writes parentheses for that node only if v has an outgoing edge with label \$ because in this case the path from the root to v corresponds to a suffix of some y_i. Moreover, leaves whose incoming edge has label \$ are ignored.

References

1. Aho, A.V., Corasick, M.: Efficient string matching: an aid to bibliographic search. Commun. ACM **18**(6), 333–340 (1975)
2. Barbay, J., Claude, F., Gagie, T., Navarro, G., Nekrich, Y.: Efficient fully-compressed sequence representations. Algorithmica **69**(1), 232–268 (2014)
3. Bauer, M.J., Cox, A.J., Rosone, G.: Lightweight algorithms for constructing and inverting the BWT of string collections. Theor. Comput. Sci. **483**, 134–148 (2013)
4. Belazzougui, D.: Linear time construction of compressed text indices in compact space. In: Proceedings of 46th Annual ACM Symposium on Theory of Computing, pp. 148–193 (2014)
5. Beller, T., Gog, S., Ohlebusch, E., Schnattinger, T.: Computing the longest common prefix array based on the Burrows-Wheeler transform. J. Discret. Algorithms **18**, 22–31 (2013)
6. Beller, T., Zwerger, M., Gog, S., Ohlebusch, E.: Space-efficient construction of the Burrows-Wheeler transform. In: Kurland, O., Lewenstein, M., Porat, E. (eds.) SPIRE 2013. LNCS, vol. 8214, pp. 5–16. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02432-5_5
7. Clark, D.: Compact pat trees. Ph.D. thesis, University of Waterloo, Canada (1996)
8. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Structuring labeled trees for optimal succinctness, and beyond. In: Proceedings of 46th Annual IEEE Symposium on Foundations of Computer Science, pp. 184–193 (2005)
9. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Compressing and indexing labeled trees, with applications. J. ACM **57**(1), Article no. 4 (2009)
10. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proceedings of 41st Annual IEEE Symposium on Foundations of Computer Science, pp. 390–398 (2000)
11. Fredkin, E.: Trie memory. Commun. ACM **3**(9), 490–499 (1960)
12. Gagie, T., Navarro, G., Puglisi, S.J.: New algorithms on wavelet trees and applications to information retrieval. Theor. Comput. Sci. **426–427**, 25–41 (2012)
13. Gog, S., Beller, T., Moffat, A., Petri, M.: From Theory to practice: plug and play with succinct data structures. In: Gudmundsson, J., Katajainen, J. (eds.) SEA 2014. LNCS, vol. 8504, pp. 326–337. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07959-2_28
14. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proceedings of 14th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 841–850 (2003)
15. Jacobson, G.: Space-efficient static trees and graphs. In: Proceedings of 30th Annual IEEE Symposium on Foundations of Computer Science, pp. 549–554 (1989)
16. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Parallel external memory suffix sorting. In: Cicalese, F., Porat, E., Vaccaro, U. (eds.) CPM 2015. LNCS, vol. 9133, pp. 329–342. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19929-0_28
17. Li, H.: Fast construction of FM-index for long sequence reads. Bioinformatics **30**(22), 3274–3275 (2014)

18. Manzini, G.: XBWT tricks. In: Inenaga, S., Sadakane, K., Sakai, T. (eds.) SPIRE 2016. LNCS, vol. 9954, pp. 80–92. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46049-9_8
19. Martínez-Prietoa, M.A., Brisaboa, N., Cánovas, R., Claude, F., Navarro, G.: Practical compressed string dictionaries. Inf. Syst. **56**, 73–108 (2016)
20. Munro, J.I.: Tables. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180, pp. 37–42. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-62034-6_35
21. Ohlebusch, E.: Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction. Oldenbusch Verlag, Bremen (2013)