View metadata, citation and similar papers at core.ac.uk brought to you by CORE
provided by Archivio Istituzionale della Ricerca- Università del Piemonte Orientale

XBWT Tricks

Giovanni Manzini[1,2]

[1]Computer Science Institute, University of Eastern Piedmont, Italy
[2]IIT-CNR, Pisa, Italy

Abstract

The eXtended Burrows-Wheeler Transform (XBWT) is a data transformation introduced in [Ferragina et al, FOCS 2005] to compactly represent a labeled tree and simultaneously support navigation and path-search operations over its label structure.

A natural application of the XBWT is to store a dictionary of strings. A recent extensive experimental study [Martínez-Prieto et al, Information Systems, 2016] shows that, among the available string dictionary implementations, the XBWT is attractive because of its good tradeoff between small space usage, speed, and support for substring searches.

In this paper we further investigate the use of the XBWT for storing a string dictionary. Our first contribution is to show how to add suffix links (aka failure links) to a XBWT string dictionary. For a XBWT dictionary with n internal nodes our suffix links can be traversed in constant time and only take $2n + o(n)$ bits of space.

Our second contribution are practical construction algorithms for the XBWT, including the additional data structure supporting the traversal of suffix links. Our algorithms build on the many well engineered algorithms for Suffix Array and BWT construction and offer different tradeoffs between running time and working space.

1 Introduction

A trie [15] is a fundamental data structure to represent a set of strings. A trie with n nodes takes $\mathcal{O}(n \log n)$ bits of space and supports extremely simple and efficient algorithms to determine whether a string belongs to the set. In this paper we are interested in the "compressed" version of a trie obtained applying to it the eXtended Burrows Wheeler Transform (XBWT): a generalization of the BWT introduced in [6, 7, 8] to compactly represent an arbitrary labeled tree. The XBWT represents an n-node trie in $\mathcal{O}(n)$ bits of space still supporting constant time upward and downward navigation.

In a recent comprehensive study of string dictionaries [18], the authors show that in many applications we need to handle dictionaries whose size is larger than the available RAM. In this setting, compression is mandatory to avoid incurring the penalties of external memory access. In the same paper the authors show that, among the available string dictionary implementations, the XBWT-trie is particularly attractive because of its good tradeoff between small space usage, speed, and support for substring searches.

In this paper we present two contributions related to the XBWT-trie. Our first contribution is the observation that we can enrich the XBWT with $2n + o(n)$ additional bits in order to support suffix links. Suffix links, also known as *failure links*, are useful to speedup some search operations as in the classical Aho-Corasick algorithm [9, Sect. 3.4].

Our second contribution is related to the problem of computing the XBWT. For a set of strings $x_1, \ldots, x_k$ of total length m we can compute the XBWT-trie by first building the n-nodes (uncompressed) trie and then applying the XBWT construction algorithm from [8]. This approach takes optimal $\mathcal{O}(m+n)$ time but it may not work well in practice because trie construction may constitute a memory bottleneck. Indeed, as shown by the Suffix-Tree vs Suffix Array debate, pointer based tree structures often have very large multiplicative constants hidden in the $\mathcal{O}$ notation that in practice prevent their use for large datasets. An indirect confirmation of this state of affairs is that in [18] the authors report that they were unable to build the trie for the largest dataset due to excessive memory usage.

In this paper we take advantage of the similarities between XBWT and BWT to derive alternative algorithms for the construction of the XBWT starting from the Suffix Array or the BWT. Our motivation is that the algorithms for constructing these data structures have been widely studied and engineered so there are practical algorithms using very little working space or even designed for external memory, see [3, 4, 5, 10, 12, 13] and references therein. Our contribution is to show that given the Suffix Array or BWT we can compute the XBWT, including the data structure supporting suffix links, in $\mathcal{O}(m)$ time. Combining our algorithms with the available (and future!) Suffix Array and BWT construction algorithms we obtain a wide range of tradeoffs between running time and working space for XBWT construction.

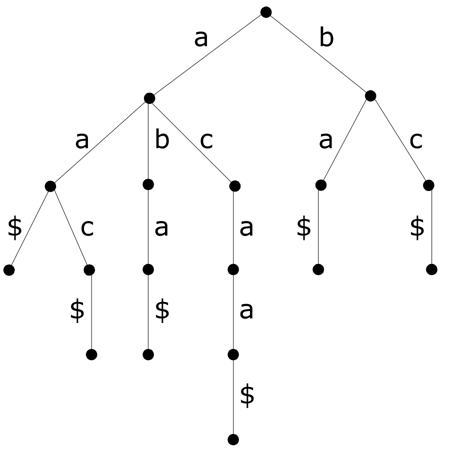
2 XBWT trie representation

Given a string $x[1,n]$ over a finite ordered alphabet Σ we write $x[i]$ to denote its i-th symbol and $x[i,j]$ to denote the substring $x[i]x[i+1]\cdots x[j]$. We write x^R to denote the string x reversed $x[n]\cdots x[1]$. We write $x \preceq y$ ($x \prec y$) to denote that x is lexicographically (strictly) smaller than y. As usual we assume that if x is a prefix of y then $x \prec y$. Throughout the paper we use the notation $\mathsf{rank}_c(x,i)$ to denote the number of occurrences of c in $x[1,i]$, and $\mathsf{select}_c(x,j)$ to denote the position of the j-th c in x.

Tries [15] are a fundamental data structure for representing a set of k distinct strings $x_1, x_2, \ldots, x_k$. A trie efficiently supports the two basic dictionary operations: $\mathsf{locate}(s)$ returning i if $s = x_i$ for some $i \in [1,k]$ or 0 otherwise, and $\mathsf{extract}(i)$ returning the string x_i given an index $i \in [1,k]$. In addition, it supports the operation $\mathsf{locatePrefix}(s)$ returning the strings which are prefixed by s [18]. To simplify the algorithms, and ensure that no string is the prefix of another one, it is customary to add a special symbol $\$ \notin \Sigma$ at the end of each string x_i. A trie for the set of strings $\{\mathsf{aa}, \mathsf{acaa}, \mathsf{ba}, \mathsf{aba}, \mathsf{aac}, \mathsf{bc}\}$ is shown in Figure 1.

The eXtended Burrows-Wheeler Transform is a generalization of the BWT designed to compactly represent a labeled tree. We now show how to compute the XBWT of a trie T and obtain two arrays L and Last that compactly represent T. Our description of the XBWT is slightly different (simpler) from the one in [6, 8] that takes as input an arbitrary labeled tree.

To each *internal* trie node w we associate the string λ_w obtained by concatenating the symbols in the arcs in the upward path from w to the root of T. Hence, if node w has depth d its associated string has length d. If T has n internal nodes we have n strings overall. Let $\Pi[1,n]$ denote the array containing the above set of n strings sorted lexicographically. Note that $\Pi[1]$ is always the empty string corresponding to the root of T.



Last	L	Π
0	a	ϵ
1	b	
0	a	a
0	b	
1	c	
0	$	aa
1	c	
1	$	aaca
1	$	ab
1	$	aba
1	a	aca
0	a	b
1	c	
1	a	ba
1	a	ca
1	$	caa
1	$	cb

Figure 1: A trie and its XBWT representation (the arrays Last and L). The array Π is not stored in the XBWT even if navigation algorithms use it to identify internal nodes.

For $i = 1, \ldots, n$ let L_i denote the set of symbols in the arcs exiting from the trie node corresponding to $\Pi[i]$. We do not require that the symbols in L_i are in any particular order, but since T is a trie they are distinct. We define the array L as the concatenation of the arrays $L_1, \ldots, L_n$. Clearly if T has n' nodes, then L has $n' - 1$ elements: one for each trie edge. By construction L contains $n-1$ symbols from Σ and $n' - n$ occurrences of \$. To keep an explicit representation of the intervals $L_1, \ldots, L_n$ we define a binary array $\mathsf{Last}[1, n']$ such that $\mathsf{Last}[i] = \mathbf{1}$ iff $L[i]$ is the last symbol of some interval L_j. Hence Last contains exactly n **1**'s. See Figure 1 for a complete example.

If $L[i] \neq \$$ belongs to the interval L_j then $L[i]$ naturally corresponds to the internal trie node reachable from the node corresponding to $\Pi[j]$ following the arc labeled $L[i]$. Such a node corresponds to the entry $\Pi[i']$ such that $\Pi[i'] = L[i]\Pi[j]$. In other words, there is a bijection between the symbols in L different from \$ and the entries in Π different from the empty string. For historical reasons this bijection is called the LF-map, and we call $LF(i)$ the index in Π of the entry corresponding to $L[i]$. Hence, LF is defined by the relation

$$\Pi[LF(i)] = L[i]\Pi[j]$$

for every i, j with $L[i] \in L_j$ and $L[i] \neq \$$. The following results are a simple restatement of Properties 1–3 in [8] using the notation of this paper.

Lemma 1 (Order preserving property)**.** *For every pair of indices i, k such that $L[i] \neq \$$, $L[k] \neq \$$, it is*

$$\begin{aligned} L[i] < L[k] &\quad\Longrightarrow\quad LF(i) < LF(k), \\ L[i] = L[k] &\quad\Longrightarrow\quad LF(i) < LF(k) \;\Leftrightarrow\; i < k. \qquad \square \end{aligned}$$

For any symbol $c \in \Sigma$ let $C(c)$ denote the index of the first position in Π containing a path starting with symbol c. Lemma 1 makes it possible to compute LF and its inverse

LF^{-1} using rank and select operations. In turn, the LF map makes it possible to navigate the XBWT-trie, that is to move from the entry in Π representing a trie node to the entries representing its children and parent.

Lemma 2 (Downward navigation)**.** *Let $c = L[i] \neq \$$. Then*

$$LF(i) = C[c] + \mathsf{rank}_c(L, i-1).$$

As a consequence, if node w corresponds to $\Pi[j]$ and has a child with label c, then such child corresponds to entry $\Pi[j']$ with

$$j' = C[c] + \mathsf{rank}_c(L, \mathsf{select}_{\mathbf{1}}(\mathsf{Last}, j)). \qquad \square$$

Lemma 3 (Upward navigation)**.** *For $i > 1$ let c denote the first symbol of path $\Pi[i]$. Then*

$$LF^{-1}(i) = \mathsf{select}_c(L, 1 + i - C[c]).$$

As a consequence, if node w corresponds to the non empty path $\Pi[j]$ whose first character is c, the parent of w' corresponds to the entry $\Pi[j']$ with

$$j' = 1 + \mathsf{rank}_{\mathbf{1}}(\mathsf{Last}, LF^{-1}(j) - 1) = 1 + \mathsf{rank}_{\mathbf{1}}(\mathsf{Last}, \mathsf{select}_c(L, 1 + j - C[c])). \qquad \square$$

Using downward (resp. upward) navigation we can implement the locate (resp. $\mathsf{extract}$) trie operation. As observed in [18] it is convenient to take as the ID of x_i the rank in L of the \$ occurrence that we reach starting from the root and following x_i's symbols. If we reorder the strings in reverse lexicographic order (ie so that $x_1^R \prec x_2^R \prec \cdots \prec x_k^R$) then $\mathsf{ID}(x_i) = i$.

The most common representation of the array L is a (possibly compressed) Wavelet tree. We also need a bitarray representation of Last supporting constant time rank_1, select_1 operations, and a suitable representation of the array C (possibly another bitarray). Using a balanced uncompressed Wavelet trees for L the space usage is $\mathcal{O}(n' \log(|\Sigma|))$ bits and each upward or downward step takes $\mathcal{O}(\log |\Sigma|)$ time.

3 Adding suffix links

In addition to pointers to their children and parent, trie nodes may store an additional pointer called a *suffix link*. The node corresponding to path α has a suffix link pointing to the node corresponding to the longest proper suffix of α that is also in T. Hence, if we have reached the node corresponding to the path $c_0 c_1 \cdots c_i$ the suffix link makes it possible to reach in constant time the node corresponding to path $c_j \cdots c_i$ where $j > 0$ is the smallest positive integer for which such node exists. Since the root corresponds to the empty string, a suffix link exists for all internal nodes except for the root itself.

In a XBWT-trie internal nodes are identified with their position in Π. Because of the ordering of the paths in Π, the target of the suffix link of node $\Pi[i]$ is the node $\ell < i$ such that $\Pi[\ell]$ is the longest proper *prefix* of $\Pi[i]$ which is in Π.

To emulate suffix links we build a string P of balanced parentheses of length $2n$. We write a pair of parentheses for each internal node so that the parentheses for node j enclose those for i iff $\Pi[j]$ is a prefix of $\Pi[i]$. To build P we start with an empty string and consider $\Pi[i]$ for $i = 1, \ldots, n$. When we reach $\Pi[i]$ first we write a) for every $\ell < i$ such that the closed parenthesis for $\Pi[\ell]$ has not been written and $\Pi[\ell]$ *is not* a prefix of $\Pi[i]$; then we write the (corresponding to $\Pi[i]$. After we have reached $\Pi[n]$ we write a closing parenthesis for all indices ℓ such that the closed parenthesis for $\Pi[\ell]$ has not yet been written. For example, for the XBWT of Figure 1 it is $P =$ `((((()()(())())(())(())()))`.

The following lemma shows that to find the suffix link for node $\Pi[i]$ it suffices to find the closest set of parentheses enclosing the (associated to $\Pi[i]$.

Lemma 4. *Let $1 < i \leq n$ and $\alpha = \Pi[i]$. Define $k = \mathsf{select}_{(}(P, i)$ and $j = \mathsf{enclose}(k)$. Then, the longest proper prefix of α in Π is $\alpha' = \Pi[\ell]$ with $\ell = \mathsf{rank}_{(}(P, j)$.*

Proof. First note that enclose is always defined since the pair $P[1] =$ (, $P[2n] =$) corresponding to $\Pi[1]$ encloses every other pair of parentheses.

We need to prove that $\alpha' = \Pi[\ell]$ is the longest proper prefix of α which is in Π. Since the) for $\Pi[\ell]$ is not written when we reach $\Pi[i]$, by construction $\Pi[\ell]$ is a prefix of $\Pi[i]$. To prove it is the longest prefix assume by contradiction that $\Pi[\ell']$ is also a prefix of $\Pi[i]$ and $|\Pi[\ell']| > |\Pi[\ell]|$. Because of the ordering in Π we would have $\ell < \ell' < i$. Also because of the ordering, for $i' = \ell' \ldots i$ $\Pi[i']$ would be a prefix of $\Pi[i]$. But then the parentheses for ℓ' would enclose those for i, which is a contradiction since by construction ℓ corresponds to the closest enclosing pair. □

Using the range min-max tree from [19] we can represent the balanced parenthesis sequence P in $2n + o(n)$ bits of space and support rank, select, and enclose in $\mathcal{O}(1)$ time. We have therefore established the following result.

Theorem 5. *We can add to the XBWT-trie suffix links traversable in constant time using additional $2n + o(n)$ bits.* □

Since Π only contains internal nodes, the approach described above only provides suffix links for the trie internal nodes. However, it can be extended to the trie leaves if necessary. Since the symbol \$ appears only at the end of a string, the suffix link of a leaf can only point to another leaf. Thus, we can build a subsequence Π' of Π containing only the internal nodes which have \$ among their children. It is easy to see that the parenthesis array P' build on Π' provides suffix links for the leaves.

4 Alternative construction algorithms

In this section we propose new algorithms for computing the XBWT of the trie containing the set of distinct strings $x_1, x_2, \ldots, x_k$. Our algorithms derive the XBWT from the Suffix Array or BWT of the concatenation $t = y_1\$y_2\$\cdots y_k\$$, where $y_i = x_i^R$ reversed and \$ is assumed to be lexicographically smaller than any symbol in Σ. We denote by SA, LCP and BWT respectively the Suffix Array, LCP array, and Burrows Wheeler Transform of the string t (See Figure 2 for an example). Throughout this section let m denote the length of t, ie $m = \sum_i(|x_i| + 1)$.

Let z be a string not containing the symbol \$ and such that $z\$$ is a substring of t. We denote by $[b_z, e_z]$ the maximal range of suffix array rows prefixed by $z\$$. For example, in Figure 2 for $z = \epsilon$ the maximal range is $[1, 6]$, for $z =$ aa the maximal range is $[11, 12]$, and for $z =$ ca the maximal range is $[20, 20]$.

#	SA	LCP	BWT	RCP	MR	Sorted suffixes
1	22	–	b	–	1	$
2	3	1	a	0	0	$aaca$ab$aba$caa$cb$
3	8	2	a	0	0	$ab$aba$caa$cb$
4	11	3	b	0	0	$aba$caa$cb$
5	15	1	a	0	0	$caa$cb$
6	19	2	a	0	0	$cb$
7	2	0	a	0	1	a$aaca$ab$aba$caa$cb$
8	7	3	c	1	0	a$ab$aba$caa$cb$
9	14	2	b	1	0	a$caa$cb$
10	18	3	a	1	0	a$cb$
11	1	1	$	1	1	aa$aaca$ab$aba$caa$cb$
12	17	3	c	2	0	aa$cb$
13	4	2	$	2	1	aaca$ab$aba$caa$cb$
14	9	1	$	1	1	ab$aba$caa$cb$
15	12	2	$	2	1	aba$caa$cb$
16	5	1	a	1	1	aca$ab$aba$caa$cb$
17	21	0	c	0	1	b$
18	10	1	a	1	0	b$aba$caa$cb$
19	13	1	a	1	1	ba$caa$cb$
20	6	0	a	0	1	ca$ab$aba$caa$cb$
21	16	2	$	2	1	caa$cb$
22	20	1	$	1	1	cb$

Figure 2: Suffix array, LCP array, MR array, and BWT for the concatenation $t =$ aa\$aaca\$ab\$aba\$caa\$cb\$ obtained from the set of strings aac, aa, aba, acaa, ba, bc. The arrays MR and RCP will be introduced later.

Lemma 6. *Let $[b_z, e_z]$ denote the maximal range for the string z. Then $e_z - b_z + 1$ is equal to the number of strings in $x_1, \ldots, x_k$ which have z as a prefix. In addition it is* $\mathsf{LCP}[b_z] < |z|$ *and*

$$\mathsf{LCP}[i] \geq |z| + 1 \qquad \textit{for } i = b_z + 1, \ldots, e_z.$$

Proof. By construction the rows prefixed by $z\$$ are in a bijection with the strings y_i's which have z as a suffix. Since $y_i = x_i^R$ the first part of the lemma follows. Since b_z is the first row prefixed by $z\$$ row $b_z - 1$ must be prefixed by a string lexicographically strictly smaller than $z\$$. Since \$ is the smallest symbol, row $b_z - 1$ cannot be prefixed by z. □

Lemma 7. *Let T denote the trie representing the strings $x_1, \ldots, x_k$. There is a one-to-one (bijective) correspondence between internal nodes of T and maximal row ranges of* SA. *Each node w corresponds to a maximal range containing a number of rows equals to the number of leaves in the subtree rooted at w. The correspondence is order preserving in the sense that row range $[b_y, e_y]$ precedes $[b_z, e_z]$ iff the node corresponding to the former interval precedes the node corresponding to the latter in the array Π used to define the XBWT.*

Proof. For each internal node w let λ_w denote the string obtained concatenating the symbols in the upward path from w to the root. The image of node w is the maximal row range associated to λ_w, that is, the set of SA rows prefixed by $\lambda_w\$$. As we have already observed, the number of rows in this interval is equal to the number of strings $x_1, \ldots, x_k$ which have λ_w as prefix which coincides with the number of leaves in the subtree rooted at w. The correspondence is order preserving since both in Π and in the suffix array the order is determined by the lexicographic order of λ_w. □

Lemma 8. *Let $[b, e]$ denote the maximal row range associated to the internal node w. Then, the labels on the arcs exiting from w coincide with the set of symbols in the substring $\mathsf{BWT}[b, e]$.*

Proof. Let λ_w denote the string containing the symbols in the upward path from w to the root. There is an arc with label $c \in \Sigma$ leaving w iff there is at least a string x_i prefixed by $\lambda_w^R c$. This implies $c\lambda_w$ is a prefix of y_i. If $j \in [b, e]$ is the row prefixed by $\lambda_w \$ y_{i+1} \$ \cdots y_k \$$ it is $\mathsf{BWT}[j] = c$. Viceversa, if $\mathsf{BWT}[h] = c$ for $h \in [b, e]$ then at least one y_i is prefixed by $c\lambda_w$, hence $\lambda_w^R c$ is a prefix of x_i and there must be an arc with label c exiting from node w.

Finally, there is an arc with label \$ leaving w iff $\lambda_w^R = x_h$ for some $h \in [1, k]$. But then there will be one SA row prefixed by $y_h \$ y_{h+1} \$ \cdots y_k \$$ and the corresponding BWT position will contain the symbol \$. □

From Lemma 8 we can derive a simple strategy to compute the XBWT, that is, the arrays L and Last defined in Section 2. Assume we are given a binary array MR such that $\mathsf{MR}[i] = \mathbf{1}$ iff row i is the starting position of a maximal row range (see example in Figure 2). MR encodes the maximal row ranges and by Lemma 7 each maximal row range corresponds to an element in the array Π. In Section 2 we have logically partitioned the array L into $L_1, \ldots, L_n$ where L_i contains the labels in the arcs leaving the internal node associated to $\Pi[i]$. We compute the subarrays $L_1, \ldots, L_n$ in that order. We scan the array MR starting from its first position until we find an index j_1 such that $\mathsf{MR}[j_1 + 1] = \mathbf{1}$. We know that $[1, j_1]$ is the maximal row range corresponding to $\Pi[1]$. In $\mathcal{O}(j_1)$ time we compute the set of distinct symbols in $\mathsf{BWT}[1, j_1]$ and we write them to L. By Lemma 8 we have just computed L_1 and we complete this phase by writing $\mathbf{0}^{|L_1|-1}\mathbf{1}$ to Last. Next we restart the scanning of MR until we find an index j_2 such that $\mathsf{MR}[j_2 + 1] = \mathbf{1}$. By construction $[j_1 + 1, j_2]$ is the maximal range corresponding to $\Pi[2]$ so from $\mathsf{BWT}[j_1 + 1, j_2]$ we can derive L_2 and so on. The above algorithm takes $\mathcal{O}(m)$ time and only requires the arrays BWT and MR.

The bit array MR can be derived from the SA and LCP arrays. However a faster alternative is to modify one of the algorithms computing the LCP from the SA so that, instead of the LCP, it computes the RCP (Reduced Common Prefix) array storing the lengths of the common prefix among lexicographically consecutive suffixes assuming that all instances of the \$ symbol are different. See again Figure 2 for an example.[1] The linear time LCP construction algorithms in [11, 14, 17] can all be easily modified to compute the RCP values instead of LCP values. The MR array can be computed along with the RCP array observing that $\mathsf{MR}[i] = \mathbf{0}$ iff $\mathsf{LCP}[i] > \mathsf{RCP}[i]$. The latter condition can be verified even without knowing the LCP values by testing whether $t[\mathsf{SA}[i] + \mathsf{RCP}[i]] = t[\mathsf{SA}[i-1] + \mathsf{RCP}[i]] = \$$. Indeed, the RCP array satisfies the following lemma which is an immediate consequence of Lemma 6.

[1] The RCP array coincides with the LCP array if we build the concatenation t inserting a different symbol $\$_i$ at the end of each string x_i. However, this approach is not practical since would increase significantly the size of the alphabet.

```
Compute $P$
1:  $S \leftarrow$ empty stack; $P \leftarrow$ empty string
2:  for $i = 1, \ldots m$ do
3:      if $\mathsf{MR}[i] == \mathbf{1}$ do     // beginning of maximal row range
4:          $\ell \leftarrow \mathsf{RCP}[i]$
5:          while $(\ell_{top} \geq \ell)$ do
6:              $S.pop()$     // pop if not prefix of the new string
7:          if $t[\mathsf{SA}[i_{top}] + \ell + 1] \neq \$$ do
8:              $\ell \leftarrow \ell_{top}$
9:              $S.pop()$
10:         $S.push(i, \ell)$
11: while ($S$ not empty) do
12:     $S.pop()$
```

Figure 3: Algorithm for computing the parenthesis array P given t, SA, RCP and MR. An open parenthesis is written to P at each *push* operation, and a closed parenthesis at each *pop* operation. (i_{top}, ℓ_{top}) represents the pair currently at the top of the stack.

Lemma 9. *Let $[b_z, e_z]$ denote the maximal range for the string $z \in \Pi$. It is*

$$\mathsf{RCP}[b_z + 1] = \mathsf{RCP}[b_z + 2] = \cdots = \mathsf{RCP}[e_z] = |z|$$

and $\mathsf{RCP}[b_z] = \mathsf{lcp}(z, z')$ where z' is the string immediately preceding z in the Π array. □

Note that computing the RCP array is faster than computing the LCP array (the common prefixes are shorter) and its storage takes less space since each entry takes at most $\lceil \log(\max_i |x_i|) \rceil$ bits.

We have established that with a single scan of the BWT and MR array we can compute the arrays L and Last. We now show that using the RCP array we can also compute the parenthesis string P that supports suffix links emulation as described in Section 3. The algorithm for computing P is described in Figure 3. To prove its correctness we first establish the following Lemma.

Lemma 10. *In the algorithm of Fig. 3 let $(i_1, \ell_1), (i_2, \ell_2), \ldots, (i_h, \ell_h)$ denote the pairs stored in the stack at any given moment, and let $z_1, z_2, \ldots, z_h$ denote the corresponding strings, i.e. z_j corresponds to the maximal row range $[i_j, e_j]$. Then, for $i = 2, \ldots, h$ we have that z_{i-1} is a proper prefix of z_i and $|z_{i-1}| = \ell_i$.*

Proof. Initially the stack is empty so the hypothesis is true. Assume now the stack $(i_1, \ell_1), (i_2, \ell_2), \ldots, (i_h, \ell_h)$ satisfies the hypothesis and we have reached position i which is the beginning of the next maximal row range which corresponds to the string z. Note that i_h is the starting point of the immediately preceding row range. Hence, setting $\ell = \mathsf{RCP}[i]$ we have $\ell = \mathsf{lcp}(z_h, z)$. In addition, for $j < h$ since z_j is a prefix of z_h it is $\mathsf{lcp}(z_j, z) = \min(\ell, |z_j|)$. Clearly if $\ell_j \geq \ell$, z_j cannot be a prefix of z since

$$|z_j| > |z_{j-1}| = \ell_j \geq \ell \geq \mathsf{lcp}(z_j, z)$$

so it is correct to remove (i_j, ℓ_j) from the stack at Line 6. If $\ell_j < \ell$ then z_j is a prefix of z iff $\ell = |z_j|$ which is the condition tested at Line 7. If this is the case we push (i, ℓ) to the stack and the invariant is maintained. If z_j is not a prefix of z then z_{j-1} certainly is, since it is a proper prefix of $z_j[1,\ell] = z[1,\ell]$, and we add to the stack (i, ℓ_j) after having removed (i_j, ℓ_j) thus maintaining the invariant. □

Theorem 11. *The algorithm of Figure 3 correctly computes the array P in $\mathcal{O}(m)$ time.*

Proof. Because of the order preserving correspondence between maximal row ranges and paths in Π, scanning the array MR is equivalent to scanning the array Π. Lemma 10 ensures that we write an open parenthesis for each path $\Pi[i]$ and that the corresponding closed parenthesis is written immediately before the opening parenthesis of the first path $\Pi[h]$ with $h > i$ such that $\Pi[i]$ is not a prefix of $\Pi[h]$. This is exactly how P is defined in Section 3 and the correctness follows.

To see that the running time is $\mathcal{O}(m)$ observe that in addition to the outer loop we only have push and pop operations on the stack. Since we push one pair (i, ℓ) for each $\mathbf{1}$ in MR, and once popped from the stack pairs are discarded, the overall time is $\mathcal{O}(m)$. □

For the construction of P, in addition to the input arrays, the algorithm needs extra storage only for the stack. Since the values in the stack are strictly increasing, it uses at most $\mathcal{O}(\ell \log \ell)$ bits where $\ell = \max_i \mathsf{RCP}[i]$. Summing up, we are able to compute the XBWT with simple sequential scans using the SA and LCP (actually RCP) arrays. Since there are many well engineered algorithms for computing the SA and LCP array, we believe our solution is the most practical choice when the working space is not an issue. Indeed, its working space is dominated by the space required for the storage and computation of the SA which is still $\mathcal{O}(m \log m)$ bits but in practice it could be much less than the space required for storing a pointer based representation of the trie T.

We now describe an alternative XBWT construction algorithm that only uses the BWT of the string $t = y_1\$y_2\$ \cdots y_k\$$. Since the BWT takes $m \log |\Sigma|$ bits and can be computed using $o(m \log m)$ bits of working space, our algorithm provides new time/space trade-offs for XBWT construction. In addition, our algorithm works without modification if the BWT of t is replaced by the Multi String BWT [10] of $\{x_1, \ldots, x_k\}$. Although BWT algorithms have been studied for a longer time, Multi String BWT algorithms are potentially faster and have recently received much attention, see [1, 10, 16] and references therein.

The idea of our algorithm is to compute the MR array emulating a depth first visit of the trie T using the BWT. Since each internal trie node corresponds to a maximal row range, the visit will give us all maximal row ranges, ie, the bit array MR. Our solution is inspired by the algorithm in [2] that computes the LCP array emulating a breadth first visit on the suffix trie using the BWT.

Assuming the BWT is stored in a balanced Wavelet Tree we can use the algorithm $\mathsf{getInterval}$ from [2] to compute, given the maximal row range corresponding to an internal node w, the maximal row ranges corresponding to w's children. This computation takes $\mathcal{O}(d \log |\Sigma|)$ time, where d is the number of w's children. Using $\mathsf{getInterval}$, the computation of the MR array can be done by the algorithm in Figure 4 whose running time is $\mathcal{O}(m + n \log |\Sigma|)$ where n is the number of internal trie nodes. The working space of the algorithm, in addition to the BWT and MR arrays, is dominated by the stack for the depth first visit which takes $(\max_i |x_i|)|\Sigma|$ words. After the computation of MR, the arrays L and Last can be obtained in $\mathcal{O}(m + |\Sigma|)$ time as described above.

To compute also the parenthesis array P we use the following approach. Our starting point is the observation that the algorithm in Figure 3 only uses the RCP values for the

```
    Compute MR (lightweight)
1:  for $i = 1, \dots m$
2:      $\mathsf{MR}[i] \leftarrow \mathbf{0}$    // Clear the MR array
3:  df_visit$(1, k, \epsilon)$

    df_visit$(b_z, e_z, z)$
1:  $\mathsf{MR}[e_z] \leftarrow \mathbf{1}$    // mark the endpoint of the maximal row range
2:  foreach $c \neq \$$ in $\mathsf{BWT}[b_z, e_z]$ do
3:      $[b_{cz}, e_{cz}] \leftarrow$ maximal row range for $cz$
4:      df_visit$(b_{cz}, e_{cz}, cz)$
```

Figure 4: Algorithm for computing the MR array given the BWT.

```
     Compute $P$ (lightweight)
 1:  $S \leftarrow$ empty stack; $P \leftarrow$ empty string
 2:  for $i = 1, \dots n$ do
 3:      $\ell \leftarrow \mathsf{RCP}'[i]$
 4:      while $(\ell_{top} \geq \ell)$ do
 5:          $S.pop()$    // pop if not prefix of the new one
 6:      if $\mathsf{LEN}'[i_{top}] \neq \ell$ do
 7:          $\ell \leftarrow \ell_{top}$
 8:          $S.pop()$
 9:      $S.push(i, \ell)$
10:  while ($S$ not empty) do
11:      $S.pop()$
```

Figure 5: Algorithm for computing the parenthesis array P given RCP' and LEN'.

entries i such that $\mathsf{MR}[i] = \mathbf{1}$. In addition, the SA is only used at Line 7 to check if the string that prefixes row i_{top} is a prefix of the string that prefixes row i. This property can be tested also by checking if the length of the string at i_{top} is equal to $\mathsf{RCP}[i]$.

This observation suggests that after the computation of MR we count the number of $\mathbf{1}$'s in it: this gives us the number n of internal trie nodes. Then, we allocate two length-n arrays RCP' and LEN' where we store the RCP and the length of the entries in Π with $\mathsf{MR}[i] = \mathbf{1}$. These arrays take $\mathcal{O}(n \log(\max_i |x_i|))$ bits and can be computed in $\mathcal{O}(n \log |\Sigma|)$ time using a straightforward modification of the LCP construction algorithm from [2]. Using RCP' and LCP' we can compute the parenthesis array P using the algorithm of Figure 5 which is derived from the one in Figure 3 but has a simpler structure since, instead of scanning MR skipping the $\mathbf{0}$ entries, it scans directly RCP' and LEN'.

5 Concluding remarks

With the advent of applications that use very large string dictionaries the XBWT-trie becomes a valid alternative for their storage. In this paper we have presented two contributions that can increase the practical appeal of this data structure. We believe there are other improvements to the original XBWT-trie design that can make this data structure even more appealing to practitioners. For example, it is relatively simple to support the contraction of unary paths. The computation of the XBWT also deserves further investigations: we have shown how to compute it from the SA or the BWT but we are currently working on the design of efficient and lightweight direct construction algorithms.

References

[1] Bauer, M.J., Cox, A.J., Rosone, G.: Lightweight algorithms for constructing and inverting the BWT of string collections. *Theor. Comput. Sci.* **483** (2013) 134–148

[2] Beller, T., Gog, S., Ohlebusch, E., Schnattinger, T.: Computing the longest common prefix array based on the Burrows-Wheeler transform. *J. Discrete Algorithms* **18** (2013) 22–31

[3] Beller, T., Zwerger, M., Gog, S., Ohlebusch, E.: Space-efficient construction of the Burrows-Wheeler transform. In: *SPIRE.* Volume 8214 of *Lecture Notes in Computer Science.*, Springer (2013) 5–16

[4] Crochemore, M., Grossi, R., Kärkkäinen, J., Landau, G.M.: Computing the Burrows-Wheeler transform in place and in small space. *J. Discrete Algorithms* **32** (2015) 44–52

[5] Ferragina, P., Gagie, T., Manzini, G.: Lightweight data indexing and compression in external memory. *Algorithmica* (2011)

[6] Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Structuring labeled trees for optimal succinctness, and beyond. In: *Proc. 46th IEEE Symposium on Foundations of Computer Science (FOCS).* (2005) 184–193

[7] Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Compressing and searching XML data via two zips. In: *Proc. 15th International World Wide Web Conference (WWW).* (2006) 751–760

[8] Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Compressing and indexing labeled trees, with applications. *J. ACM* **57** (2009)

[9] Gusfield, D.: *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press (1997)

[10] Holt, J., McMillan, L.: Constructing Burrows-Wheeler transforms of large string collections via merging. In: *BCB*, ACM (2014) 464–471

[11] Kärkkäinen, J., Manzini, G., Puglisi, S.: Permuted longest-common-prefix array. In: *Proc. 20th Symposium on Combinatorial Pattern Matching (CPM)*, Springer-Verlag, LNCS n. 5577 (2009) 181–192

[12] Kärkkäinen, J., Kempa, D.: Engineering a lightweight external memory suffix array construction algorithm. In: *ICABD.* Volume 1146 of *CEUR Workshop Proceedings.*, CEUR-WS.org (2014) 53–60

[13] Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Parallel external memory suffix sorting. In: *CPM.* Volume 9133 of *Lecture Notes in Computer Science.*, Springer (2015) 329–342

[14] Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: *Proc. 12th Symposium on Combinatorial Pattern Matching (CPM '01)*, Springer-Verlag LNCS n. 2089 (2001) 181–192

[15] Knuth, D.E.: *Sorting and Searching.* Second edn. Volume 3 of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, USA (1998)

[16] Li, H.: Fast construction of FM-index for long sequence reads. *Bioinformatics* **30** (2014) 3274–3275

[17] Manzini, G.: Two space saving tricks for linear time LCP computation. In: *Proc. of 9th Scandinavian Workshop on Algorithm Theory (SWAT '04)*, Springer-Verlag LNCS n. 3111 (2004) 372–383

[18] Martínez-Prieto, M.A., Brisaboa, N.R., Cánovas, R., Claude, F., Navarro, G.: Practical compressed string dictionaries. *Inf. Syst.* **56** (2016) 73–108

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