Journal of Discrete Algorithms ••• (••••) •••–•••

Contents lists available at SciVerse ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

Computing the Burrows–Wheeler transform of a string and its reverse in parallel

Enno Ohlebusch [a,*], Timo Beller [a], Mohamed I. Abouelhoda [b,c]

[a] *Institute of Theoretical Computer Science, University of Ulm, 89069 Ulm, Germany*
[b] *Center for Informatics Sciences, Nile University, Giza, Egypt*
[c] *Faculty of Engineering, Cairo University, Giza, Egypt*

A R T I C L E   I N F O

Article history:
Available online xxxx

Keywords:
Burrows–Wheeler transform
Reverse string
Wavelet tree
Parallel computing

A B S T R A C T

The contribution of this article is twofold. First, we provide new theoretical insights into the relationship between a string and its reverse: If the Burrows–Wheeler transform (BWT) of a string has been computed by sorting its suffixes, then the BWT, the suffix array, and the longest common prefix array of the reverse string can be derived from it without suffix sorting. Furthermore, we show that the longest common prefix arrays of a string and its reverse are permutations of each other. Second, we provide a parallel algorithm that, given the BWT of a string, computes the BWT of its reverse much faster than all known (parallel) suffix sorting algorithms. Some bioinformatics applications will benefit from this.

© 2013 Elsevier B.V. All rights reserved.

1. Introduction

The Burrows–Wheeler transform [2] is used in many lossless data compression programs, of which the best known is Julian Seward's bzip2. Moreover, it is the basis of FM-indexes that support backward search [4]. In some bioinformatics applications, one needs both the Burrows–Wheeler transform BWT of a string S and the Burrows–Wheeler transform BWT^{rev} of the reverse string S^{rev}. A prime example is short-read mapping (finding the positions of short sequences of 25–150 base pairs—called reads—within a long reference sequence); see e.g. [5,22] for overview articles. Short-read mappers are, among others, Bowtie [15], BWA [16], and 2BWT [14]. All three software-tools use BWT and BWT^{rev}: BWA to reduce the search space, Bowtie to search from both ends, and 2BWT to search bidirectionally. In the prediction of RNA-coding genes [20], it is also advantageous to be able to search in forward and backward direction. This bidirectional search requires BWT as support for backward search and BWT^{rev} as support for forward search. Another example is *de novo* sequence assembly based on pairwise overlaps between sequence reads. Simpson and Durbin [21] showed how an assembly string graph can be efficiently constructed using all pairs of exact suffix–prefix overlaps between reads. (Välimäki et al. [23] provide techniques to find all pairs of approximate suffix–prefix overlaps.) To compute overlaps between reverse complemented reads, they build an FM-index for the set of reads *and* an FM-index for the set of *reversed* reads.

The Burrows–Wheeler transform BWT of a string S is usually computed by sorting all suffixes of S (hence the suffix array of S is known). Of course, the Burrows–Wheeler transform BWT^{rev} of the reverse string S^{rev} can be obtained in the same fashion. However, because of the strong relationship between a string and its reverse, it is quite natural to ask whether BWT^{rev} can be directly derived from BWT—without sorting the suffixes of S^{rev}. In this article, we prove that this is indeed the case. More precisely, we give an algorithm for this task that has $O(n \log \sigma)$ worst-case time complexity. Interestingly, essentially the same algorithm can be applied to obtain the Burrows–Wheeler transform of the reverse complement of a

* Corresponding author.
E-mail addresses: Enno.Ohlebusch@uni-ulm.de (E. Ohlebusch), Timo.Beller@uni-ulm.de (T. Beller), mabouelhoda@yahoo.com (M.I. Abouelhoda).

1570-8667/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.jda.2013.06.002

i	SA	BWT	$S_{\mathsf{SA}[i]}$
1	10	g	\$
2	3	t	$aataatg\$$
3	6	t	$aatg\$$
4	4	a	$ataatg\$$
5	7	a	$atg\$$
6	1	\$	$ctaataatg\$$
7	9	t	$g\$$
8	2	c	$taataatg\$$
9	5	a	$taatg\$$
10	8	a	$tg\$$

i	SA	BWT	$S^{rev}_{\mathsf{SA}[i]}$	LCP	LF
1	$\underline{10}$	c	\$	$-\underline{1}$	6
2	3	t	$aataatc\$$	$\underline{0}$	8
3	6	t	$aatc\$$	3	9
4	4	a	$ataatc\$$	$\underline{1}$	2
5	7	a	$atc\$$	2	3
6	$\underline{9}$	t	$c\$$	$\underline{0}$	10
7	$\underline{1}$	\$	$gtaataatc\$$	$\underline{0}$	1
8	$\underline{2}$	g	$taataatc\$$	$\underline{0}$	7
9	$\underline{5}$	a	$taatc\$$	$\underline{4}$	4
10	$\underline{8}$	a	$tc\$$	$\underline{1}$	5

Fig. 1. Left-hand side: suffix array SA and BWT of the string $S = ctaataatg\$$ (the input). Right-hand side: Burrows–Wheeler transform of $S^{rev} = gtaataatc\$$ (the output). The computation of the LF-mapping, the suffix array, and the lcp-array of S^{rev} will be explained in Section 4, Section 5, and Section 6, respectively.

DNA sequence. Furthermore, we show that LF^{rev}—the last-to-first mapping for BWT^{rev}—can be computed along with BWT^{rev}. The suffix array SA^{rev} and the lcp-array LCP^{rev} of S^{rev} can also be obtained with our new algorithm, but only partially. Fortunately, the arrays can be completed by means of the LF^{rev}-mapping. In contrast to suffix sorting, our new algorithm is easy to parallelize. Experiments show that it is faster than all known (parallel) suffix sorting algorithms.

This article is organized as follows. In the next section, we introduce some preliminaries. In Section 3, we present the main result of this article: an algorithm that takes a string S, its suffix array SA, and its Burrows–Wheeler transform BWT as input and returns BWT^{rev} in $O(n \log \sigma)$ time. That LF^{rev} can be computed along with BWT^{rev} is shown in Section 4. In Section 5, we show that our algorithm can be used to fill SA^{rev} partially and how the array can be completed. In Section 6, it is shown that our algorithm is able to compute all irreducible lcp-values of LCP^{rev} and that the remaining reducible lcp-values can easily be derived from them. We further prove that LCP^{rev} is a permutation of LCP. Section 7 outlines how the algorithm can be parallelized. In Section 8, we present our experimental results, and in the concluding section we summarize our results.

A preliminary version of this article appeared in [18].

2. Preliminaries

Let Σ be an ordered alphabet of size σ whose smallest element is the so-called sentinel character \$. In the following, S is a string of length n over Σ having the sentinel character at the end (and nowhere else). For $1 \leqslant i \leqslant n$, $S[i]$ denotes the *character at position* i in S. For $i \leqslant j$, $S[i..j]$ denotes the *substring* of S starting with the character at position i and ending with the character at position j. Furthermore, S_i denotes the i-th suffix $S[i..n]$ of S.

The *suffix array* SA of the string S is an array of integers in the range 1 to n specifying the lexicographic ordering of the n suffixes of the string S, that is, it satisfies $S_{\mathsf{SA}[1]} < S_{\mathsf{SA}[2]} < \cdots < S_{\mathsf{SA}[n]}$; see Fig. 1 for an example. We refer to the overview article [19] for construction algorithms (some of which have linear run time). In the following, ISA denotes the inverse of the permutation SA.

The *suffix tree* ST for S is a compact trie storing the suffixes of S: for any leaf i, the concatenation of the edge labels on the path from the root to leaf i exactly spells out the suffix S_i. In the following, we denote an internal node α in ST by $\overline{\omega}$, where ω is the concatenation of the edge labels on the path from the root to α. A pointer from an internal node $\overline{c\omega}$ to the internal node $\overline{\omega}$ is called a *suffix link*; see [10] for details.

The Burrows and Wheeler transform [2] converts a string S into the string $\mathsf{BWT}[1..n]$ defined by $\mathsf{BWT}[i] = S[\mathsf{SA}[i] - 1]$ for all i with $\mathsf{SA}[i] \neq 1$ and $\mathsf{BWT}[i] = \$$ otherwise; see Fig. 1. The permutation LF, defined by $LF(i) = \mathsf{ISA}[\mathsf{SA}[i] - 1]$ for all i with $\mathsf{SA}[i] \neq 1$ and $LF(i) = 1$ otherwise, is called LF-mapping. The LF-mapping can be implemented by $LF(i) = C[c] + Occ(c, i)$, where $c = \mathsf{BWT}[i]$, $C[c]$ is the overall number of characters in S that are strictly smaller than c, and $Occ(c, i)$ is the number of occurrences of the character c in $\mathsf{BWT}[1..i]$.

The *lcp-array* of S is an array LCP so that $\mathsf{LCP}[1] = -1$ and $\mathsf{LCP}[i] = |\mathsf{lcp}(S_{\mathsf{SA}[i-1]}, S_{\mathsf{SA}[i]})|$ for $2 \leqslant i \leqslant n$, where $\mathsf{lcp}(u, v)$ denotes the longest common prefix between two strings u and v; see Fig. 1. It can be computed in linear time from the suffix array and its inverse; see [13,17,12,8]. A value $\mathsf{LCP}[i]$ is called *reducible* if $\mathsf{BWT}[i] = \mathsf{BWT}[i-1]$; otherwise it is *irreducible*.

Ferragina and Manzini [4] showed that it is possible to search a pattern backwards, character by character, in the suffix array SA of string S, without storing SA. Let $c \in \Sigma$ and ω be a substring of S. Given the ω-interval $[i..j]$ in the suffix array SA of S (i.e., ω is a prefix of $S_{\mathsf{SA}[k]}$ for all $i \leqslant k \leqslant j$, but ω is not a prefix of any other suffix of S), $backwardSearch(c, [i..j])$ returns the $c\omega$-interval $[C[c] + Occ(c, i-1) + 1..C[c] + Occ(c, j)]$. In our example of Fig. 1, $backwardSearch(a, [2..5])$ returns the aa-interval $[2..3]$.

The *wavelet tree* introduced by Grossi et al. [9] supports one backward search step in $O(\log \sigma)$ time. To explain this data structure, we may view the ordered alphabet Σ as an array of size σ so that the characters appear in ascending order in the array $\Sigma[1..\sigma]$, i.e., $\Sigma[1] = \$ < \Sigma[2] < \cdots < \Sigma[\sigma]$. We say that an interval $[l..r]$ is an *alphabet interval*, if it is a subinterval of $[1..\sigma]$. For an alphabet interval $[l..r]$, the string $\mathsf{BWT}^{[l..r]}$ is obtained from the Burrows–Wheeler transformed string BWT of S

by deleting all characters in BWT that do not belong to the sub-alphabet $\Sigma[l..r]$ of $\Sigma[1..\sigma]$. The wavelet tree of the string BWT over the alphabet $\Sigma[1..\sigma]$ is a balanced binary search tree defined as follows. Each node v of the tree corresponds to a string $\mathsf{BWT}^{[l..r]}$, where $[l..r]$ is an alphabet interval. The root of the tree corresponds to the string $\mathsf{BWT} = \mathsf{BWT}^{[1..\sigma]}$. If $l = r$, then v has no children. Otherwise, v has two children: its left child corresponds to the string $\mathsf{BWT}^{[l..m]}$ and its right child corresponds to the string $\mathsf{BWT}^{[m+1..r]}$, where $m = \lfloor \frac{l+r}{2} \rfloor$. In this case, v stores a bit vector $B^{[l..r]}$ whose i-th entry is 0 if the i-th character in $\mathsf{BWT}^{[l..r]}$ belongs to the sub-alphabet $\Sigma[l..m]$ and 1 if it belongs to the sub-alphabet $\Sigma[m+1..r]$. To put it differently, an entry in the bit vector is 0 if the corresponding character belongs to the left subtree and 1 if it belongs to the right subtree. Moreover, each bit vector B in the tree is preprocessed so that the queries $rank_0(B,i)$ and $rank_1(B,i)$ can be answered in constant time, where $rank_b(B,i)$ is the number of occurrences of bit b in $B[1..i]$. Obviously, the wavelet tree has height $O(\log\sigma)$. Because in an actual implementation it suffices to store only the bit vectors, the wavelet tree requires only $n\log\sigma$ bits of space plus $o(n\log\sigma)$ bits for the data structures that support rank queries in constant time.

Algorithm 1 For an ω-interval $[i..j]$, the function call *getIntervals*$([i..j])$ returns the list of all $c\omega$-intervals, and is defined as follows.

```
getIntervals($[i..j]$)
  $list \leftarrow [\,]$
  $getIntervals'([i..j], [1..\sigma], list)$
  return $list$

getIntervals'($[i..j], [l..r], list$)
  if $l = r$ then
    $c \leftarrow \Sigma[l]$
    $add(list, [C[c]+i..C[c]+j])$
  else
    $(a_0, b_0) \leftarrow (rank_0(B^{[l..r]}, i-1), rank_0(B^{[l..r]}, j))$
    $(a_1, b_1) \leftarrow (i-1-a_0, j-b_0)$
    $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$
    if $b_0 > a_0$ then
      $getIntervals'([a_0+1..b_0], [l..m], list)$
    if $b_1 > a_1$ then
      $getIntervals'([a_1+1..b_1], [m+1..r], list)$
```

For an ω-interval $[i..j]$, the procedure *getIntervals*$([i..j])$ presented in Algorithm 1 returns the list of all $c\omega$-intervals; cf. [1,3]. More precisely, it starts with the ω-interval $[i..j]$ at the root and traverses the wavelet tree in a top-down fashion as follows. At the current node v, it uses constant time rank queries to obtain the number $b_0 - a_0$ of zeros in the bit vector of v within the current interval. If $b_0 > a_0$, then there are characters in $\mathsf{BWT}[i..j]$ that belong to the left subtree of v, and the algorithm proceeds recursively with the left child of v. Furthermore, if the number of ones is positive (i.e. if $b_1 > a_1$), then it proceeds with the right child in an analogous fashion. Clearly, if a leaf corresponding to character c is reached with current interval $[p..q]$, then $[C[c]+p..C[c]+q]$ is the $c\omega$ interval. In this way, Algorithm 1 computes the list of all $c\omega$-intervals. This takes $O(k\log\sigma)$ time for a k-element list. In our example of Fig. 1, *getIntervals*$([2..5])$ returns the list $[[2..3],[8..9]]$, where $[2..3]$ is the *aa*-interval and $[8..9]$ is the *ta*-interval.

3. The Burrows–Wheeler transform of the reverse string

If we reverse the order of the characters in a string, we obtain its *reverse string*. For technical reasons, however, we assume that the sentinel symbol \$ occurs at the end of each string under consideration. For this reason, the reverse string S^{rev} of a string S that is terminated by \$ is obtained by deleting \$ from S, reversing the order of the characters, and appending \$. For example, the reverse string of $S = ctaataatg\$$ is $S^{rev} = gtaataatc\$$ (and not $\$gtaataatc$).

Given the BWT and the suffix array SA of S, Algorithm 2 recursively computes the Burrows–Wheeler transformed string BWT^{rev} of S^{rev} by the procedure call $bwtrev(1,[1..n],0)$. The string BWT^{rev} is computed in a left-to-right fashion: first $\mathsf{BWT}^{rev}[1]$, then $\mathsf{BWT}^{rev}[2]$, etc. Suppose that the algorithm has already calculated the first $k-1$ characters of BWT^{rev}. When it tries to determine $\mathsf{BWT}^{rev}[k]$, it is known that $[k..k+rb-lb]$ is the $\omega^{rev}c$-interval in SA^{rev} because the $c\omega$-interval $[lb..rb]$ in SA has been identified by backward search (with the help of the wavelet tree of BWT). At that point, the algorithm proceeds by case analysis (below, $\ell+1$ is the length of $c\omega$):

- If each occurrence of $c\omega$ in S is followed by the same character a (in particular, this is true whenever $lb = rb$), then each occurrence of $\omega^{rev}c$ is preceded by a in S^{rev}. Thus, $\mathsf{BWT}^{rev}[q] = a$ for every q with $k \leqslant q \leqslant k+rb-lb$.
- If not all the characters in $\mathsf{BWT}^{rev}[k..k+rb-lb]$ are the same, then the algorithm recursively determines the lexicographic order of the suffixes in the $\omega^{rev}c$-interval $[k..k+rb-lb]$ as far as it is needed.

Algorithm 2 Procedure $bwtrev(k, [i..j], \ell)$ uses the wavelet tree of the BWT, the suffix array SA, and S. The call $bwtrev(1, [1..n], 0)$ computes BWT^{rev}.

```
bwtrev($k$, $[i..j]$, $\ell$)
  $list \leftarrow getIntervals([i..j])$ /* intervals in increasing lexicographic order */
  while $list$ not empty do
    $[lb..rb] \leftarrow head(list)$
    if $lb = rb$ or $S[\mathsf{SA}[lb] + \ell + 1] = S[\mathsf{SA}[rb] + \ell + 1]$ then
      $pos \leftarrow \mathsf{SA}[lb] + \ell + 1$
      if $lb = rb$ then
        if $pos > n$ then
          $pos \leftarrow pos - n$
        $\mathsf{SA}^{rev}[k] \leftarrow n - pos + 1$ /* this will be explained in Section 5 */
      $c \leftarrow S[pos]$
      for $q \leftarrow k$ to $k + rb - lb$ do
        $\mathsf{BWT}^{rev}[q] \leftarrow c$
        $count[c] \leftarrow count[c] + 1$ /* this will be explained in Section 4 */
        $LF^{rev}[q] \leftarrow count[c]$ /* this will be explained in Section 4 */
    else
      bwtrev($k$, $[lb..rb]$, $\ell + 1$)
    $k \leftarrow k + rb - lb + 1$
    if $list$ not empty then
      $\mathsf{LCP}^{rev}[k] \leftarrow \ell$ /* this will be explained in Section 6 */
```

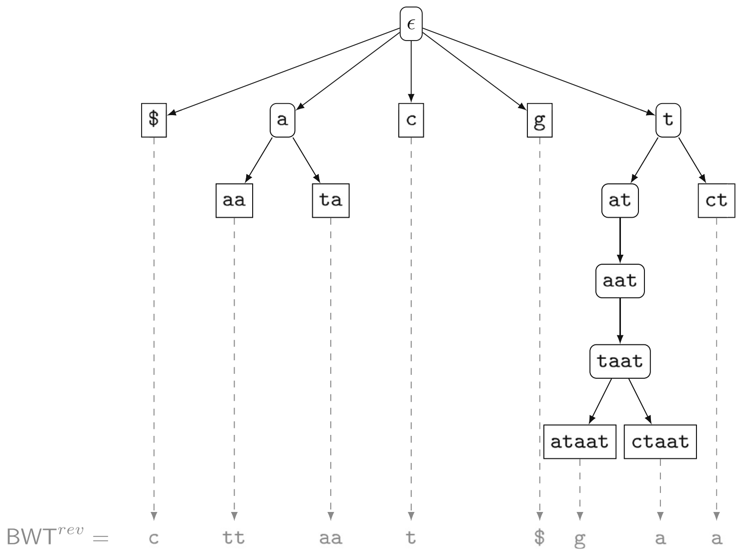

Fig. 2. The recursion tree of Algorithm 2 applied to the BWT of $S = ctaataatg\$$: for each internal node there is a recursive call. E.g., the recursive call with the aat-interval as parameter determines the characters of BWT^{rev} in the taa-interval of SA^{rev}.

We exemplify the algorithm by applying it to $S = ctaataatg\$$. The procedure call $getIntervals([1..10])$ returns the list $[[1..1], [2..5], [6..6], [7..7], [8..10]]$, where $[1..1]$ is the \$-interval, $[2..5]$ is the a-interval, and so on; cf. Fig. 1. Then, the first interval $[lb..rb] = [1..1]$ is taken from the list (note that $head(list)$ removes the first element of $list$ and returns it). Because $lb = rb$, the algorithm computes $pos = \mathsf{SA}[1] + 0 + 1 = 11$. Furthermore, since $pos > n = 10$, it assigns the character $S[11 - 10] = S[1] = c$ to $\mathsf{BWT}^{rev}[1]$ and increments k (so the new value of k is 2). Now the interval $[lb..rb] = [2..5]$ is taken from the list. Because $S[\mathsf{SA}[lb] + \ell + 1] = S[\mathsf{SA}[2] + 0 + 1] = S[4] = a \neq t = S[8] = S[\mathsf{SA}[5] + 0 + 1] = S[\mathsf{SA}[rb] + \ell + 1]$, Algorithm 2 recursively calls $bwtrev(2, [2..5], 1)$. The procedure call $getIntervals([2..5])$ returns the list $[[2..3], [8..9]]$, where $[2..3]$ is the aa-interval and $[8..9]$ is the ta-interval. Then, the first interval $[lb..rb] = [2..3]$ is taken from the list. In this case, $S[\mathsf{SA}[lb] + \ell + 1] = S[\mathsf{SA}[2] + 1 + 1] = S[5] = t = t = S[8] = S[\mathsf{SA}[3] + 1 + 1] = S[\mathsf{SA}[rb] + \ell + 1]$. Thus, t is assigned to both $\mathsf{BWT}^{rev}[2]$ and $\mathsf{BWT}^{rev}[3]$. Now the algorithm continues with the new value $k = 2 + 3 - 2 + 1 = 4$. Fig. 2 shows the recursion tree of Algorithm 2.

In essence, the correctness of Algorithm 2 is a consequence of the following lemma. Moreover, the lemma allows us to parallelize the algorithm.

Lemma 3.1. *Let $[i..j]$ be the ω-interval for some substring ω of S, and let k be the left boundary of the ω^{rev}-interval in* SA^{rev}. *If $[lb_1..rb_1], \ldots, [lb_m..rb_m]$ are the intervals in list $= getIntervals([i..j])$ corresponding to the strings $c_1\omega, \ldots, c_m\omega$, where $c_1 < \cdots < c_m$, then the intervals $[s_1..e_1], \ldots, [s_m..e_m]$ in SA^{rev} corresponding to the strings $\omega^{rev}c_1, \ldots, \omega^{rev}c_m$ satisfy $s_q = k + \sum_{p=1}^{q-1}(rb_p - lb_p + 1)$ and $e_q = s_q + (rb_q - lb_q)$, where $1 \leqslant q \leqslant m$.*

Proof. We prove the lemma by finite induction on q. In the base case $q = 1$. Because c_1 is the smallest character in Σ for which the $c_1\omega$-interval is non-empty, the suffixes of S^{rev} that have $\omega^{rev}c_1$ as a prefix are lexicographically smaller than those suffixes of S^{rev} that have $\omega^{rev}c_p$, $2 \leqslant p \leqslant m$, as a prefix. Hence $s_1 = k$. Moreover, it follows from the fact that the $\omega^{rev}c_1$-interval has size $rb_1 - lb_1 + 1$ that the $\omega^{rev}c_1$-interval is the interval $[k..k + (rb_1 - lb_1)]$. For the inductive step suppose that the $\omega^{rev}c_{q-1}$-interval $[s_{q-1}..e_{q-1}]$ satisfies $s_{q-1} = k + \sum_{p=1}^{q-2}(rb_p - lb_p + 1)$ and $e_{q-1} = s_{q-1} + (rb_{q-1} - lb_{q-1})$. Because c_q is the q-th smallest character in Σ for which the $c_q\omega$-interval is non-empty, the suffixes of S^{rev} that have $\omega^{rev}c_q$ as a prefix are lexicographically larger than the suffixes of S^{rev} that have $\omega^{rev}c_p$ as a prefix, where $1 \leqslant p \leqslant q - 1$. It follows that the $\omega^{rev}c_q$-interval $[s_q..e_q]$ satisfies $s_q = e_{q-1} + 1 = k + \sum_{p=1}^{q-1}(rb_p - lb_p + 1)$ and $e_q = s_q + (rb_q - lb_q)$. □

Theorem 3.2. *Algorithm 2 correctly computes* BWT^{rev}.

Proof. We prove the theorem by induction on k. Suppose Algorithm 2 is applied to the ω-interval $[i..j]$ for some ℓ-length substring ω of S, and let the $c\omega$-interval $[lb..rb]$ be the interval dealt with in the current execution of the while-loop. According to the inductive hypothesis, $\mathsf{BWT}^{rev}[1..k-1]$ has been computed correctly. Moreover, by Lemma 3.1, $[k..k+rb-lb]$ is the $\omega^{rev}c$-interval in SA^{rev}. We further proceed by a case-by-case analysis.

- If $lb = rb$, then $c\omega$ occurs exactly once in S and it is the length $\ell+1$ prefix of suffix $S_{\mathsf{SA}[lb]}$. In this case, the suffix of S^{rev} that has $\omega^{rev}c$ as a prefix is the k-th lexicographically smallest suffix of S^{rev}. The character $\mathsf{BWT}^{rev}[k]$ is $S[\mathsf{SA}[lb] + \ell + 1]$ because this is the character that immediately follows the prefix $S[\mathsf{SA}[lb]..\mathsf{SA}[lb] + \ell] = c\omega$ of suffix $S_{\mathsf{SA}[lb]}$.
- If $lb \neq rb$ and $S[\mathsf{SA}[lb] + \ell + 1] = S[\mathsf{SA}[rb] + \ell + 1]$, then each occurrence of $c\omega$ in S is followed by the same character $a = S[\mathsf{SA}[lb] + \ell + 1]$. Thus, each suffix of S^{rev} in the $\omega^{rev}c$-interval $[k..k + rb - lb]$ is preceded by a. Consequently, $\mathsf{BWT}^{rev}[q] = a$ for every q with $k \leqslant q \leqslant k + rb - lb$. So in this case, it is not necessary to know the exact lexicographic order of the suffixes in the $\omega^{rev}c$-interval.
- If $lb \neq rb$ and $S[\mathsf{SA}[lb] + \ell + 1] \neq S[\mathsf{SA}[rb] + \ell + 1]$, then not all the characters in $\mathsf{BWT}^{rev}[k..k + rb - lb]$ are the same and thus the recursive call $bwtrev(k, [lb..rb], \ell + 1)$ determines the lexicographic order of the suffixes in the $\omega^{rev}c$-interval $[k..k + rb - lb]$ as far as it is needed. □

Next, we analyze the worst-case time complexity of Algorithm 2.

Lemma 3.3. *The procedure bwtrev is executed with the parameters $(k, [i..j], \ell)$, where $[i..j]$ is the ω-interval for some substring ω of S, if and only if $\overline{\omega}$ is an internal node in the suffix tree ST of S.*

Proof. We use induction on ℓ. There is just one procedure call with $\ell = 0$, namely $bwtrev(1, [1..n], 0)$. The interval $[1..n]$ is the ε-interval, where ε denotes the empty string. Clearly, the node $\overline{\varepsilon}$ is the root node of the suffix tree ST, and the root is an internal node. According to the inductive hypothesis, the procedure *bwtrev* is executed with the parameters $(k, [i..j], \ell)$, where $[i..j]$ is the ω-interval for some substring ω of S, if and only if $\overline{\omega}$ is an internal node in the suffix tree ST of S. For the inductive step, assume that $[lb..rb]$ is one of the intervals returned by the procedure call $getIntervals([i..j])$, say the $c\omega$-interval. We prove that there is a recursive procedure call $bwtrev(k', [lb..rb], \ell + 1)$ if and only if $\overline{c\omega}$ is an internal node of ST. It is readily verified that $\overline{c\omega}$ is an internal node of ST if and only if the $c\omega$-interval contains two different suffixes of S, one having $c\omega a$ as a prefix and one having $c\omega b$ as a prefix, where a and b are different characters from Σ. Again, we proceed by case analysis:

- If $lb = rb$, then $c\omega$ occurs exactly once in S. Hence $\overline{c\omega}$ is not an internal node of ST. Note that Algorithm 2 does not invoke a recursive call to *bwtrev*.
- If $lb \neq rb$ and $S[\mathsf{SA}[lb] + \ell + 1] = S[\mathsf{SA}[rb] + \ell + 1]$, then each occurrence of $c\omega$ in S is followed by the same character. Again, $\overline{c\omega}$ is not an internal node of ST and Algorithm 2 does not invoke a recursive call to *bwtrev*.
- If $a = S[\mathsf{SA}[lb] + \ell + 1] \neq S[\mathsf{SA}[rb] + \ell + 1] = b$, then $\overline{c\omega}$ is an internal node of ST and Algorithm 2 invokes the recursive call $bwtrev(k', [lb..rb], \ell + 1)$. □

Lemma 3.3 implies that the recursion tree of Algorithm 2 coincides with the suffix link tree SLT of S, defined as follows.

Definition 3.4. The suffix link tree SLT of a suffix tree ST has a node $\underline{\omega}$ for each internal node $\overline{\omega}$ of ST. For each suffix link from $\overline{c\omega}$ to $\overline{\omega}$ in ST, there is an edge $\underline{\omega} \to \underline{c\omega}$ in SLT.

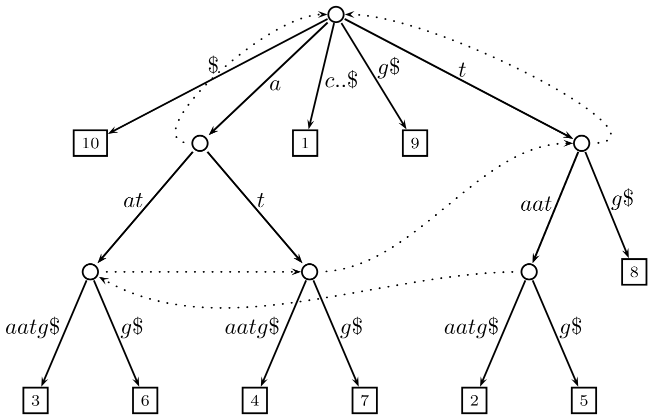

Fig. 3. The suffix tree for $S = ctaataatg\$$.

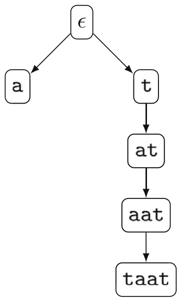

Fig. 4. The suffix link tree for $S = ctaataatg\$$.

Fig. 3 shows the suffix tree of the string $S = ctaataatg\$$ and Fig. 4 shows the corresponding suffix link tree.

That the recursion tree of Algorithm 2 coincides with the suffix link tree SLT of S can be seen as follows. If the execution of $bwtrev(k, [i..j], \ell)$ invokes the recursive call $bwtrev(k', [lb..rb], \ell + 1)$, where $[i..j]$ is the ω-interval and $[lb..rb]$ is the $c\omega$-interval, then there is a suffix link from node $\overline{c\omega}$ to node $\overline{\omega}$ because both are internal nodes in the suffix tree of S.

Theorem 3.5. *Algorithm 2 has a worst-case time complexity of $O(n \log \sigma)$.*

Proof. According to Lemma 3.3, there are as many recursive calls to the procedure *bwtrev* as there are internal nodes in the suffix tree ST of S. Because ST has n leaves and each internal node in ST is branching, the number of internal nodes is at most $n - 1$. We use an amortized analysis to show that the overall number of intervals returned by calls to the procedure *getIntervals* is bounded by $2n - 1$. Let L denote the concatenation of all lists returned by procedure calls to *getIntervals*. For each element $[lb..rb]$ of L, either at least one entry of BWT^{rev} is filled in, or there is a recursive call to the procedure *bwtrev*. It follows that L has at most $2n - 1$ elements because BWT^{rev} has n entries and there are at most $n - 1$ recursive calls to the procedure *bwtrev*. It is a consequence of this amortized analysis that the overall time taken by all procedure calls to *getIntervals* is $O(n \log \sigma)$ because a procedure call to *getIntervals* that returns a k-element list takes $O(k \log \sigma)$ time. Clearly, the theorem follows from this fact. $\square$

The Burrows–Wheeler transform of the reverse complement of a DNA-sequence can also be computed by Algorithm 2. One just has to change the order in which intervals are generated by the procedure *getIntervals*. Recall that the reverse complement of a DNA-sequence S is obtained by reversing S and then replacing each nucleotide by its Watson–Crick complement (a is replaced with t and vice versa; c is replaced with g and vice versa). For example, the reverse complement of $ctaataatg$ is $cattattag$; see Fig. 5 for the corresponding suffix arrays and Burrows–Wheeler transforms. Up to now, a ϕ-interval was generated before an ω-interval if and only if $\phi^{rev} <_{lex} \omega^{rev}$, where $<_{lex}$ is the lexicographic order induced by the order $\$ < a < c < g < t$ on the alphabet Σ. In the computation of the Burrows–Wheeler transform of the reverse complement of a DNA-sequence, a ϕ-interval must be generated before an ω-interval if and only if $\phi^{rev} <_{lex'} \omega^{rev}$, where $<_{lex'}$ is the lexicographic order induced by the order $\$ < t < g < c < a$. Moreover, instead of assigning the nucleotide $S[\mathsf{SA}[lb] + \ell + 1]$ to a position in BWT^{rev}, its Watson–Crick complement must be assigned. The recursion tree of our example can be found in Fig. 6.

i	SA	BWT	$S_{\mathsf{SA}[i]}$
1	10	g	\$
2	3	t	$aataatg\$$
3	6	t	$aatg\$$
4	4	a	$ataatg\$$
5	7	a	$atg\$$
6	1	\$	$ctaataatg\$$
7	9	t	$g\$$
8	2	c	$taataatg\$$
9	5	a	$taatg\$$
10	8	a	$tg\$$

i	SA^{rev}	BWT^{rev}	$S^{rev}_{\mathsf{SA}^{rev}[i]}$
1	10	g	\$
2	8	t	$ag\$$
3	5	t	$attag\$$
4	2	c	$attattag\$$
5	1	\$	$cattattag\$$
6	9	a	$g\$$
7	7	t	$tag\$$
8	4	t	$tattag\$$
9	6	a	$ttag\$$
10	3	a	$ttattag\$$

Fig. 5. Left-hand side: suffix array SA and BWT of the string $S = ctaataatg\$$. Right-hand side: SA^{rev} and BWT^{rev} of the reverse complement $cattattag\$$ of S.

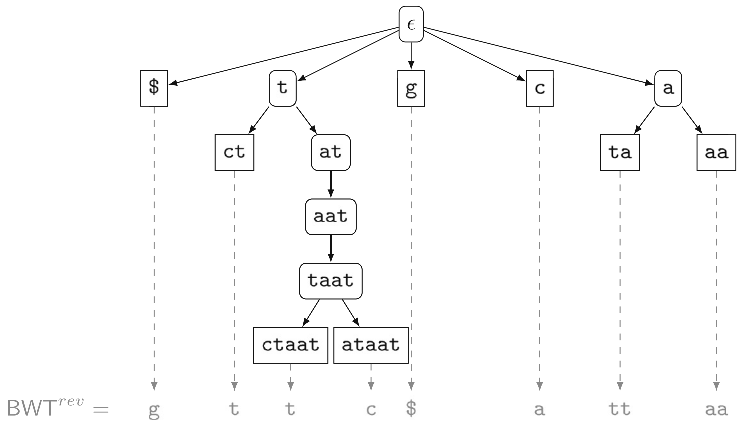

Fig. 6. The recursion tree of Algorithm 2 with modified procedure *getIntervals* applied to the BWT of the string $S = ctaataatg\$$, calculating the Burrows–Wheeler transform of the reverse complement $cattattag\$$.

Algorithm 3 Computing LF from BWT and the C-array.

```
for all $c \in \Sigma$ do
    $count[c] \leftarrow C[c]$
for $i \leftarrow 1$ to $n$ do
    $c \leftarrow \mathsf{BWT}[i]$
    $count[c] \leftarrow count[c] + 1$
    $LF[i] \leftarrow count[c]$
```

4. The *LF*-mapping of the reversed string

In this section, we show that Algorithm 2 can be used to compute LF^{rev} along with BWT^{rev}. Let us recall that the inverse of the Burrows–Wheeler transformation of a string[1] is based on the LF-mapping; see [2]. This mapping is called last-to-first mapping because in the matrix of the sorted cyclic shifts of the original string it maps the last column L (which coincides with the string BWT) to the first column F (where $F[i] = S[\mathsf{SA}[i]]$ for each i with $1 \leqslant i \leqslant n$). The LF- mapping has the following key property: If $L[i] = c$ is the k-th occurrence of character c in $L = \mathsf{BWT}$, then $LF(i) = j$ is the index so that $F[j]$ is the k-th occurrence of c in F. With this key property, the LF-mapping can be calculated by Algorithm 3 from BWT and the C-array (recall that if we consider all characters in Σ that are alphabetically smaller than c, then $C[c]$ is the overall number of their occurrences in BWT). If Algorithm 3 finds the k-th occurrence of character c at index i in BWT, then the k-th occurrence of character c in F appears at index $count[c] = C[c] + k$. Hence $LF[i] = count[c]$.

From this perspective, it is immediately clear that Algorithm 2 can be used to compute LF^{rev} along with BWT^{rev}. To be precise, if the *count* array is initialized as in Algorithm 3, then the procedure call $bwtrev(1, [1..n], 0)$ computes BWT^{rev} as well as LF^{rev}. Note that the *count* arrays of S, S^{rev}, BWT, and BWT^{rev} coincide because these are permutations of each other.

[1] In our context, the string under consideration is S^{rev}.

5. The suffix array of the reversed string

Algorithm 2 recursively computes the *whole* Burrows–Wheeler transformed string BWT^{rev} of S^{rev} but it cannot be used to calculate the *whole* suffix array SA^{rev}. This is because a S^{rev}-value can be assigned in only one of the two base cases of the recursion.

- In the base case $lb = rb$, the character $c = S[pos]$ is assigned to $\mathsf{BWT}^{rev}[k]$; see Algorithm 2. If $c \neq \$$, then this occurrence of c appears at position $n - pos$ in S^{rev}. Thus, $\mathsf{SA}^{rev}[k] = n - pos + 1$. If $c = \$$, then $pos = n$ and \$ also appears at position n in S^{rev}. Again, $\mathsf{SA}^{rev}[k] = n - pos + 1$.
- If $lb \neq rb$ and $S[\mathsf{SA}[lb] + \ell + 1] = S[\mathsf{SA}[rb] + \ell + 1]$, then all the characters in $\mathsf{BWT}^{rev}[k..k + rb - lb]$ are the same and the algorithm does not determine the lexicographic order of the suffixes. In this case, the values in $\mathsf{SA}^{rev}[k..k + rb - lb]$ remain unknown.

It follows as a consequence that Algorithm 2 fills the suffix array SA^{rev} only partially; in Fig. 1 the computed entries are underlined. Nevertheless, partial information is better than no information.

Next, we explain how a partially filled suffix array SA of a string[2] can be completed. It is well-known that the LF-mapping can not only be used to recover the original string from the BWT but also its suffix array. This is a direct consequence of the equation

$$\mathsf{SA}[i] = \mathsf{SA}\big[LF[i]\big] + 1$$

Algorithm 4 shows pseudo-code that computes the suffix array by means of the LF-mapping; see e.g. [17] for a similar algorithm.

Algorithm 4 Computing SA from LF.

```
$j \leftarrow n$ /* $\mathsf{SA}[1] = n$ */
$k \leftarrow LF[1]$
**while** $LF[k] \neq 1$ **do**
    $j \leftarrow j - 1$
    $\mathsf{SA}[k] \leftarrow j$
    $k \leftarrow LF[k]$
```

In principle, the pseudo-code on the left-hand side of Algorithm 5 proceeds as Algorithm 4 does. In a left-to-right scan of the SA array, whenever it finds a defined SA entry (an entry that has already been computed), it follows LF-pointers and fills in $\mathsf{SA}[i]$ entries that have not been computed yet until another defined SA entry is reached.

Alternatively, one can use the pseudo-code on the right-hand side of Algorithm 5 to complete the SA array. This algorithm also scans the SA array from left to right, but this time it ignores defined entries. Instead, whenever it finds an undefined entry $\mathsf{SA}[i]$, it follows LF-pointers until an index k is reached with $\mathsf{SA}[k] \neq \perp$, and it stores the sequence $i, LF(i), \ldots, LF^q(i)$ on a stack, where $k = LF^{q+1}(i)$. Clearly, if $\mathsf{SA}[k] = j$, then the SA-value at index $LF^q(i)$—the topmost element of the stack—is $\mathsf{SA}[LF^q(i)] = j + 1$. After $LF^q(i)$ has been popped from the stack, the subsequent values $\mathsf{SA}[LF^{q-1}(i)], \ldots, \mathsf{SA}[LF(i)], \mathsf{SA}[i]$ are obtained similarly.

Algorithm 5 Given a partial suffix array, these procedures compute the whole suffix array SA.

```
**for** $i \leftarrow 1$ **to** $n$ **do**
    **if** $\mathsf{SA}[i] \neq \perp$ **then**
        $j \leftarrow \mathsf{SA}[i]$
        $k \leftarrow LF[i]$
        **while** $\mathsf{SA}[k] = \perp$ **do**
            $j \leftarrow j - 1$
            $\mathsf{SA}[k] \leftarrow j$
            $k \leftarrow LF[k]$
```

```
initialize an empty stack
**for** $i \leftarrow 1$ **to** $n$ **do**
    $k \leftarrow i$
    **while** $\mathsf{SA}[k] = \perp$ **do**
        *push*$(k)$
        $k \leftarrow LF[k]$
    $j \leftarrow \mathsf{SA}[k]$
    **while** stack is not empty **do**
        $j \leftarrow j + 1$
        $\mathsf{SA}[pop()] \leftarrow j$
```

[2] In our context, the string under consideration is S^{rev}.

6. The lcp-array of the reversed string

In fact, Algorithm 2 can also be used to compute LCP^{rev}. This can be seen as follows. Suppose Algorithm 2 is applied to the ω-interval $[i..j]$ for some ℓ-length substring ω of S, and let k be the left boundary of the ω^{rev}-interval in SA^{rev}. It is a consequence of Lemma 3.1 that the procedure call $bwtrev(k, [i..j], \ell)$ correctly computes the boundaries $[s_q..e_q]$ of the $\omega^{rev}c_q$-intervals in SA^{rev}, where $c_1, \dots, c_m$ are the characters for which $c_q\omega$ is a substring of S ($1 \leqslant q \leqslant m$). By the conditional statement "**if** *list* not empty **then** $\mathsf{LCP}^{rev}[k] \leftarrow \ell$ ", Algorithm 2 assigns the value ℓ at each index $s_2, \dots, s_m$ (but not at index s_1). This is correct, i.e., $\mathsf{LCP}^{rev}[s_q] = \ell$ for $2 \leqslant q \leqslant m$, because $\omega^{rev}c_{q-1}$ is a prefix of the suffix at index e_{q-1} and $\omega^{rev}c_q$ is a prefix of the suffix at index $s_q = e_{q-1} + 1$.

Thus, whenever Algorithm 2 fills an entry in the lcp-array LCP^{rev}, it assigns the correct value. However, the algorithm does not fill LCP^{rev} completely; in Fig. 1 the computed entries are underlined. This is because whenever Algorithm 2 detects a $c\omega$-interval $[lb..rb]$ in the list returned by $getIntervals([i..j])$ with $lb \neq rb$ and $S[\mathsf{SA}[lb] + \ell + 1] = S[\mathsf{SA}[rb] + \ell + 1]$, then it does *not* determine the lexicographic ordering of the suffixes in the $\omega^{rev}c$-interval $[s..e]$. Instead, it fills $\mathsf{BWT}^{rev}[s..e]$ with a's because each occurrence of $c\omega$ in S is followed by the same character $a = S[\mathsf{SA}[lb] + \ell + 1]$. Consequently, if an entry $\mathsf{LCP}^{rev}[p]$ is not filled by Algorithm 2, then $\mathsf{BWT}^{rev}[p-1] = \mathsf{BWT}^{rev}[p]$. Hence $\mathsf{LCP}^{rev}[p]$ is reducible. To sum up, Algorithm 2 computes all irreducible lcp-values of LCP^{rev} (and possibly some reducible values).

Algorithm 6 Given a partial LCP array that contains at least all irreducible LCP-values, this procedure computes the whole LCP array.

```
initialize an empty stack
for $i \leftarrow 1$ to $n$ do
  $k \leftarrow i$
  while $\mathsf{LCP}[k] = \bot$ do
    $push(k)$
    $k \leftarrow LF[k]$
  $\ell \leftarrow \mathsf{LCP}[k]$
  while stack is not empty do
    $\ell \leftarrow \ell - 1$
    $\mathsf{LCP}[pop()] \leftarrow \ell$
```

Algorithm 6 shows pseudo-code for the computation of the whole LCP array of a string,[3] provided that all irreducible LCP-values are already stored in the LCP array. The key property utilized by Algorithm 6 is that a *reducible* value $\mathsf{LCP}[i]$ can be computed by the equation (see [17, Lemma 1] and [12, Lemma 4])

$$\mathsf{LCP}[i] = \mathsf{LCP}\big[LF[i]\big] - 1$$

Because Algorithm 6 is very similar to the pseudo-code on the right-hand side of Algorithm 5, we do not explain it in detail.

In the remainder of this section, we prove a strong relationship between LCP and LCP^{rev}.

Lemma 6.1. *The longest common prefix array* LCP^{rev} *of* S^{rev} *is a permutation of the longest common prefix array* LCP *of* S.

Proof. We show that each lcp-value occurs as often in LCP as in LCP^{rev}. Let $\ell \in \{1, \dots, n\}$ and define the set M_ℓ by $M_\ell = \{\omega \mid \omega$ is an ℓ-length substring but not a suffix of $S\}$. We count how many entries in the array LCP are smaller than ℓ. There are ℓ proper suffixes of S having a length $\leqslant \ell$. For each such suffix $S_{\mathsf{SA}[k]}$ we have $\mathsf{LCP}[k] < \ell$. Any other suffix has a length greater than ℓ and hence its ℓ-length prefix belongs to M_ℓ. Let $\omega \in M_\ell$ and let $[i..j]$ be the ω-interval. Clearly, for all k with $i < k \leqslant j$, we have $\mathsf{LCP}[k] \geqslant \ell$ because the suffixes $S_{\mathsf{SA}[k-1]}$ and $S_{\mathsf{SA}[k]}$ share the prefix ω. By contrast, $\mathsf{LCP}[i] < \ell$ because ω is not a prefix of $S_{\mathsf{SA}[i-1]}$. Thus, there are $|M_\ell|$ many entries in the array LCP satisfying $\mathsf{LCP}[k] < \ell$ and $|S_{\mathsf{SA}[k]}| > \ell$. In total, the array LCP has $|M_\ell| + \ell$ many entries that are smaller than ℓ. Analogously, there are $|M_{\ell+1}| + \ell + 1$ many entries in the array LCP that are smaller than $\ell + 1$, where $\ell \in \{1, \dots, n-1\}$. Consequently, the lcp-value ℓ occurs $(|M_{\ell+1}| + \ell + 1) - (|M_\ell| + \ell) = |M_{\ell+1}| - |M_\ell| + 1$ times in the LCP array. By the same argument, the lcp-value ℓ occurs $|M^{rev}_{\ell+1}| - |M^{rev}_\ell| + 1$ many times in the array LCP^{rev}, where $M^{rev}_\ell = \{\omega \mid \omega$ is an ℓ-length substring but not a suffix of $S^{rev}\}$. Now the lemma follows from the equality $|M_\ell| = |M^{rev}_\ell|$, which is true because $\omega \in M_\ell$ if and only if $\omega^{rev} \in M^{rev}_\ell$. □

Fig. 7 illustrates the proof of Lemma 6.1. The proper suffixes of S with length $\leqslant 2$ occur at the indices 1 and 4 in the (conceptual) suffix array, so LCP[1] and LCP[4] are smaller than 2. Furthermore, we have $M_2 = \{at, ca, gc, tc, tg\}$ and the corresponding entries in the LCP array at the indices 2, 5, 7, 9, and 10 are also smaller than 2. So there are $|M_2| + 2 = 7$ entries of the LCP array that are smaller than 2. Since $M_3 = \{atc, atg, cat, gca, tgc\}$, there are $|M_3| + 3 = 8$ entries of the

[3] In our context, the string under consideration is S^{rev}.

i	LCP	$S_{\mathsf{SA}[i]}$
1	−1	$
2	0	*atc*$
3	2	*atgcatc*$
4	0	*c*$
5	1	*catc*$
6	3	*catgcatc*$
7	0	*gcatc*$
8	4	*gcatgcatc*$
9	0	*tc*$
10	1	*tgcatc*$

i	LCP^{rev}	$S^{rev}_{\mathsf{SA}^{rev}[i]}$
1	−1	$
2	0	*acg*$
3	3	*acgtacg*$
4	0	*cg*$
5	2	*cgtacg*$
6	1	*ctacgtacg*$
7	0	*g*$
8	1	*gtacg*$
9	0	*tacg*$
10	4	*tacgtacg*$

Fig. 7. The lcp-arrays of $S = gcatgcatc\$$ and $S^{rev} = ctacgtacg\$$.

LCP array that are smaller than 3. We conclude that the value 2 occurs $8 - 7 = 1$ times in the LCP array. By the same argument, it occurs only once in LCP^{rev}. Note that $M_2^{rev} = \{ac, cg, ct, gt, ta\}$ and $M_3^{rev} = \{acg, cgt, cta, gta, tac\}$.

7. Parallelization

According to Lemma 3.1, the following two properties hold in Algorithm 2:

- Any two tuples $(k, [lb..rb], \ell)$ and $(k', [lb'..rb'], \ell')$ with $[k..k + rb - lb] \cap [k'..k' + rb' - lb'] = \emptyset$ are independent of each other and can thus be processed in parallel.
- Each entry of the array BWT^{rev} is accessed only once. The same is true for the arrays SA^{rev} and LCP^{rev}.

These properties can be exploited to develop an efficient parallel version of Algorithm 2. For a shared-memory multi-core system, the parallel algorithm works as in Algorithm 7. Given P processors, the procedure *expandIntervals* generates $K = m \cdot P$ pairwise independent tuples that cover the whole interval $[1..n]$. These tuples are stored in a queue Q. The algorithm then invokes Algorithm 2 to process the tuples in parallel. To have a better load balancing and to keep all processors busy, K is chosen as a multiple of P.

It should be pointed out that LF^{rev} is computed sequentially with Algorithm 3. This is because $LF[i]$ depends on $\mathsf{BWT}^{rev}[1..i]$ and in a parallel implementation one cannot assure that $\mathsf{BWT}^{rev}[1..i]$ is available when $LF^{rev}[i]$ is computed.

Algorithm 7 Parallel procedure *bwtrev*.

```
$Q \leftarrow expandIntervals([1..n])$
for each $(k, [lb..rb], \ell)$ in $Q$ do
    $bwtrev(k, [lb..rb], \ell)$
```

Algorithms 5 and 6 that complete the arrays SA^{rev} and LCP^{rev} are so similar that they can be parallelized by the same strategy. Here, we focus solely on the algorithm on the left-hand side of Algorithm 5; the other algorithms are dealt with similarly. Recall from Section 5 that SA^{rev} is partially filled and that Algorithm 5 (left-hand side) uses known entries as starting points to compute unknown entries. So if we distribute the known entries over different processors, then the array can be completed in parallel. This idea can be implemented by parallelizing the for-loop of Algorithm 5.

8. Implementation details and experimental results

We implemented Algorithm 2 using Simon Gog's [7] library sdsl (http://github.com/simongog/sdsl) and conducted experiments on a multi-core machine of 40 processors (Intel Xeon processors with 2.0 GHz; L1 Cache = 32 K, L2 Cache = 256 K, and L3 Cache = 18 M) and a total of 256 GB RAM. The operating system was CentOS release 6.2. All programs were compiled with gcc/g++ (version 4.4.6) using the -msse4.2 and -O3 optimization options. The parallelization was done by OpenMP directives.

Table 1 shows the datasets used in our experiments and their characteristics. Chromosome 1 of the human genome and chromosome 22 of the mouse genome are available at http://www.ncbi.nlm.nih.gov/ and the other test files can be found at http://code.google.com/p/libdivsufsort/wiki/SACA_Benchmarks. Table 1 also shows the times for constructing the arrays SA^{rev}, BWT^{rev}, and LCP^{rev} sequentially. The suffix array SA^{rev} was constructed with Yuta Mori's library libdivsufsort (http://code.google.com/p/libdivsufsort), which contains one of the fastest sequential suffix array construction algorithms. BWT^{rev} was derived in linear time from SA^{rev} using the equation $\mathsf{BWT}^{rev}[i] = S^{rev}[\mathsf{SA}^{rev}[i] - 1]$ for all i with $\mathsf{SA}^{rev}[i] \neq 1$ and $\mathsf{BWT}^{rev}[i] = \$$ otherwise. Furthermore, we used the Φ-algorithm [12] to compute the LCP^{rev} array directly because it is one of the fastest lcp-array construction algorithms.

Table 1
The first three columns show the data, the underlying alphabet size, and the size of the data (in MB). The columns titled divsuf, divbwt, and divlcp contain the times (in seconds) for constructing the suffix array, the Burrows–Wheeler transform, and the lcp-array of the reverse string as described in the text. The last column divAll shows the sum of the three times. The I/O operations for reading and writing files are not included in the CPU times reported in these columns, but the column titled IO(divAll) shows the I/O time for the construction of all tables.

Data	σ	size	divsuf	divbwt	divlcp	divAll	IO(divAll)
E.coli	4	4.6	0.6	0.3	0.3	1.2	0.2
mouse chr22	4	34	7.8	1.7	2.1	11.6	2.5
human chr1	4	200	44.4	11.3	14.4	70.1	9.1
swissprot	20	22	5.6	1.6	1.2	8.4	0.9
dickens	100	10.2	1.3	0.3	0.5	2.1	0.3

Table 2
Running times in seconds for the datasets of Table 1 using three versions of Algorithm 2. bwt-rev v1 is an implementation of Algorithm 2 without modification, bwt-rev v2 incorporates the special treatment of the case $rb - lb = 1$, and bwt-rev v3 extends this treatment to the case $rb - lb < 100$.

E.coli							
Tool	P1	P2	P4	P8	P16	P32	P40
bwt-rev v1	1.67	1.04	0.66	0.49	0.40	0.37	0.37
bwt-rev v2	1.30	0.78	0.56	0.44	0.44	0.39	0.36
bwt-rev v3	1.34	0.68	0.35	0.18	0.09	0.06	0.05
mouse Chr22							
Tool	P1	P2	P4	P8	P16	P32	P40
bwt-rev v1	18.38	10.59	6.42	4.21	3.57	2.69	2.65
bwt-rev v2	14.52	9.09	5.40	4.06	2.85	1.87	1.25
bwt-rev v3	10.87	5.72	2.83	1.42	0.72	0.45	0.39
human Chr1							
Tool	P1	P2	P4	P8	P16	P32	P40
bwt-rev v1	132.5	82.4	50.59	34.74	25.37	21.45	21.40
bwt-rev v2	101.42	66.53	41.13	28.55	25.59	23.03	22.54
bwt-rev v3	69.2	36.45	18.02	9.19	4.77	2.84	2.50
swissprot							
Tool	P1	P2	P4	P8	P16	P32	P40
bwt-rev v1	13.76	8.05	4.76	3.00	2.47	2.25	2.14
bwt-rev v2	8.97	5.47	3.56	2.29	1.96	1.83	1.75
bwt-rev v3	6.47	2.89	1.42	0.72	0.36	0.25	0.23
dickens							
Tool	P1	P2	P4	P8	P16	P32	P40
bwt-rev v1	5.65	3.08	1.82	1.63	1.04	1.07	0.9
bwt-rev v2	3.98	2.54	1.48	1.1	0.92	0.92	0.7
bwt-rev v3	2.47	1.22	0.61	0.31	0.25	0.24	0.23

The parallel implementation of Algorithm 2 is called bwt-rev v1 (v1 stands for version 1) in Table 2. In [18], the experiments were conducted with a variant of this implementation, called bwt-rev v2 in Table 2, which handles the case $rb - lb = 1$ separately. In that case, the relative order of the corresponding two suffixes of S^{rev} is determined as follows: the substrings of S starting at the positions $\mathsf{SA}[lb] - 1$ and $\mathsf{SA}[rb] - 1$ are compared, character by character, in a right-to-left scan until a mismatch occurs. Clearly, the mismatching characters determine the order of the two suffixes. This approach causes less cache misses and does not use rank queries. As expected, it is faster in practice; see Table 2. In bwt-rev v3, we extended this optimization so that each interval $[lb..rb]$ with $rb - lb < 100$ is treated in an analogous fashion. As one can see from Table 2, in most cases bwt-rev v3 is the fastest version, and it scales well with the number of processors. That is why the following experiments were conducted with this version, simply called bwt-rev from now on.

Table 3 shows the result of an experimental comparison of our new algorithms with known algorithms. The experiments were conducted with parallel implementations of Algorithms 2, 5 (left-hand side), and 6. For a fair comparison, we used two parallel suffix sorting algorithms: the mkESA package [11] (only for DNA/protein datasets) and our own implementation (called PBS) of an improved version of the algorithm of [6]. (To the best of our knowledge, no other implementation of a parallel suffix sorting algorithm running on a multi-core architecture is available.) Once a suffix array SA^{rev} is constructed with mkESA or PBS, we derive BWT^{rev} in linear time from it and compute LCP^{rev} sequentially as explained above (the running times can be found in the columns divbwt and divlcp of Table 1). Table 3 shows the experimental results for the datasets listed in Table 1. The times for one processor $P1$ correspond to the sequential implementations of our algorithms. As can be seen in this table, our method outperforms the PBS and mkESA tools in terms of absolute running time and

Table 3
Running times (in seconds) for the datasets of Table 1 using different tools and different numbers of processors (e.g. P32 = 32 processors). The rows titled bwt-rev, sa-rev, and lcp-rev contain the times for constructing the Burrows–Wheeler transform, the suffix array, and the lcp-array of the reverse string with the parallel implementations of our algorithms. The row all-rev shows the sum of the three times. The times for the tools PBS and mkESA include the time for constructing the other arrays as described in the text (so they should be compared with the times in the row all-rev). Because mkESA can solely handle biological data, it could not be applied to the *dickens* dataset. The I/O operations for reading and writing files are excluded in these columns, but the last column shows the I/O time. The values f-sa and f-lcp are explained in the text.

E.coli

Tool	P1	P2	P4	P8	P16	P32	P40	(f-sa, f-lcp)	IO
bwt-rev	1.34	0.68	0.35	0.18	0.09	0.06	0.05		
sa-rev	0.09	0.05	0.03	0.01	0.01	0.01	0.01	(0.8, 0.9)	
lcp-rev	0.08	0.05	0.03	0.01	0.01	0.01	0.01		
all-rev	1.51	0.78	0.41	0.2	0.11	0.08	0.07		0.3
PBS	2.78	2.08	1.78	1.45	1.33	1.23	1.18		
mkESA	1.88	1.78	1.68	1.68	1.68	1.68	1.68		

mouse Chr22

Tool	P1	P2	P4	P8	P16	P32	P40	(f-sa, f-lcp)	IO
bwt-rev	10.87	5.72	2.83	1.42	0.72	0.45	0.39		
sa-rev	1.01	0.61	0.32	0.16	0.08	0.05	0.05	(0.7, 0.8)	
lcp-rev	0.78	0.47	0.30	0.14	0.07	0.04	0.04		
all-rev	12.66	6.8	3.45	1.72	0.87	0.54	0.48		3.0
PBS	34.85	20.85	14.45	10.98	9.2	8.75	8.65		
mkESA	13.95	12.1	11.35	11.35	11.35	11.35	11.35		

human Chr1

Tool	P1	P2	P4	P8	P16	P32	P40	(f-sa, f-lcp)	IO
bwt-rev	69.2	36.45	18.02	9.19	4.77	2.84	2.50		
sa-rev	5.46	3.45	1.74	0.87	0.46	0.35	0.29	(0.74, 0.84)	
lcp-rev	3.63	2.6	1.44	0.73	0.38	0.23	0.21		
all-rev	78.29	42.5	21.2	10.79	5.61	3.42	3.0		12.7
PBS	252	160	106	80	68	63	62		
mkESA	124	114	109	109	109	109	109		

swissprot

Tool	P1	P2	P4	P8	P16	P32	P40	(f-sa, f-lcp)	IO
bwt-rev	6.47	2.89	1.42	0.72	0.36	0.25	0.23		
sa-rev	0.99	0.56	0.3	0.15	0.09	0.05	0.04	(0.64, 0.75)	
lcp-rev	0.74	0.44	0.26	0.12	0.06	0.04	0.03		
all-rev	8.2	3.89	1.98	0.99	0.51	0.34	0.3		1.18
PBS	29.6	18.4	13.3	9.7	8.4	8	7.7		
mkESA	12.6	12.1	10.2	9.9	9.3	8.9	8.9		

dickens

Tool	P1	P2	P4	*P*8	P16	P32	P40	(f-sa, f-lcp)	IO
bwt-rev	2.47	1.22	0.61	0.31	0.25	0.24	0.23		
sa-rev	0.38	0.21	0.11	0.06	0.03	0.03	0.03	(0.51, 0.6)	
lcp-rev	0.23	0.15	0.08	0.04	0.02	0.01	0.01		
all-rev	3.08	1.58	0.8	0.41	0.3	0.28	0.27		0.42
PBS	15.9	12.9	11.7	10.9	9.7	9.6	9.6		

scalability. Moreover, in a comparison with the results in Table 1, we see that with just two processors our method is faster than the method based on libdivsufsort, the fastest sequential suffix sorting algorithm. Of course, it is interesting to know how many entries of the arrays SA^{rev} and LCP^{rev} are filled already by Algorithm 2. The pairs (f-sa, f-lcp) in the last column of Table 3 provide this information. In case of *E.coli*, for example, the pair $(0.8, 0.9)$ indicates that 80% of the array SA^{rev} and 90% of the array LCP^{rev} are filled by Algorithm 2. In other words, only 20% of SA^{rev} and 10% of LCP^{rev} must be completed with Algorithms 5 (left-hand side) and 6, respectively.

Table 4 compares the space consumption of the sequential method with our new parallel method. We stress, however, that memory was not an issue of the implementation. In other words, the memory use can be reduced at the cost of slower program execution. For example, the sequential method requires about 9 bytes per input character because the Φ-algorithm does. If one uses a different algorithm to compute the LCP^{rev} array, e.g. the one described in [8], then the memory requirement drops to 5 bytes per input character.

Table 4
The space consumption in bytes per input character of the sequential method divAll explained in Table 1 and of the new methods explained in Table 3.

Data	divAll	bwt-rev	sa-rev	lcp-rev	all-rev
E.coli	9.30	6.59	10.59	10.59	14.59
mouse chr22	9.04	5.82	9.82	9.82	13.82
human chr1	9.01	5.90	9.90	9.90	13.90
swissprot	9.06	6.34	10.34	10.34	14.34
dickens	9.14	6.86	10.86	10.86	14.86

9. Conclusions

In this article, we have presented an algorithm that takes a string S, its suffix array SA, and its Burrows–Wheeler transform BWT as input and computes BWT^{rev}, LF^{rev}, SA^{rev}, and LCP^{rev} in $O(n \log \sigma)$ time. Although, the last two arrays are filled only partially, they can easily be completed in $O(n)$ time with the help of LF^{rev}. In contrast to suffix sorting, our algorithm can easily be parallelized: with just two processors it outperforms the best sequential algorithm. Consequently, the algorithm is of relevance for applications that need both BWT and BWT^{rev}. A prime example in bioinformatics is short-read mapping.

References

[1] T. Beller, S. Gog, E. Ohlebusch, T. Schnattinger, Computing the longest common prefix array based on the Burrows–Wheeler transform, in: Proc. 18th International Symposium on String Processing and Information Retrieval, in: Lecture Notes in Computer Science, vol. 7024, Springer-Verlag, Berlin, 2011, pp. 197–208.
[2] M. Burrows, D.J. Wheeler, A block-sorting lossless data compression algorithm, Research Report 124, Digital Systems Research Center, 1994.
[3] J.S. Culpepper, G. Navarro, S.J. Puglisi, A. Turpin, Top-k ranked document search in general text databases, in: Proc. 18th Annual European Symposium on Algorithms, in: Lecture Notes in Computer Science, vol. 6347, Springer-Verlag, Berlin, 2010, pp. 194–205.
[4] P. Ferragina, G. Manzini, Opportunistic data structures with applications, in: Proc. IEEE Symposium on Foundations of Computer Science, 2000, pp. 390–398.
[5] P. Flick, E. Birney, Sense from sequence reads: Methods for alignment and assembly, Nature Methods 6 (11 Suppl.) (2009) S6–S12.
[6] N. Futamura, S. Aluru, S. Kurtz, Parallel suffix sorting, in: Proc. 9th International Conference on Advanced Computing and Communications, IEEE, 2001, pp. 76–81.
[7] S. Gog, Compressed suffix trees: design, construction, and applications, PhD thesis, University of Ulm, Germany, 2011.
[8] S. Gog, E. Ohlebusch, Compressed suffix trees: Efficient computation and storage of LCP-values, Journal of Experimental Algorithmics 18 (2013) 2.1:2.1–2.1:2.31, http://dx.doi.org/10.1145/2444016.2461327.
[9] R. Grossi, A. Gupta, J.S. Vitter, High-order entropy-compressed text indexes, in: Proc. 14th Annual ACM–SIAM Symposium on Discrete Algorithms, 2003, pp. 841–850.
[10] D. Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University Press, New York, 1997.
[11] R. Homann, D. Fleer, R. Giegerich, M. Rehmsmeier, mkESA: Enhanced suffix array construction tool, Bioinformatics 25 (8) (2009) 1084–1085.
[12] J. Kärkkäinen, G. Manzini, S.J. Puglisi, Permuted longest-common-prefix array, in: Proc. 20th Annual Symposium on Combinatorial Pattern Matching, in: Lecture Notes in Computer Science, vol. 5577, Springer-Verlag, Berlin, 2009, pp. 181–192.
[13] T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, in: Proc. 12th Annual Symposium on Combinatorial Pattern Matching, in: Lecture Notes in Computer Science, vol. 2089, Springer-Verlag, Berlin, 2001, pp. 181–192.
[14] T.-W. Lam, R. Li, A. Tam, S. Wong, E. Wu, S.-M. Yiu, High throughput short read alignment via bi-directional BWT, in: Proc. International Conference on Bioinformatics and Biomedicine, IEEE Computer Society, 2009, pp. 31–36.
[15] B. Langmead, C. Trapnell, M. Pop, S.L. Salzberg, Ultrafast and memory-efficient alignment of short DNA sequences to the human genome, Genome Biology 10 (R25) (2009).
[16] H. Li, R. Durbin, Fast and accurate short read alignment with Burrows–Wheeler Transform, Bioinformatics 25 (14) (2009) 1754–1760.
[17] G. Manzini, Two space saving tricks for linear time LCP array computation, in: Proc. 9th Scandinavian Workshop on Algorithm Theory, in: Lecture Notes in Computer Science, vol. 3111, Springer-Verlag, Berlin, 2004, pp. 372–383.
[18] E. Ohlebusch, T. Beller, M.I. Abouelhoda, Computing the Burrows–Wheeler transform of a string and its reverse, in: Proc. 23rd Annual Symposium on Combinatorial Pattern Matching, in: Lecture Notes in Computer Science, vol. 7354, Springer-Verlag, Berlin, 2012, pp. 243–256.
[19] S.J. Puglisi, W.F. Smyth, A. Turpin, A taxonomy of suffix array construction algorithms, ACM Computing Surveys 39 (2) (2007) 1–31.
[20] T. Schnattinger, E. Ohlebusch, S. Gog, Bidirectional search in a string with wavelet trees, in: Proc. 21st Annual Symposium on Combinatorial Pattern Matching, in: Lecture Notes in Computer Science, vol. 6129, Springer-Verlag, Berlin, 2010, pp. 40–50.
[21] J.T. Simpson, R. Durbin, Efficient construction of an assembly string graph using the FM-index, Bioinformatics 26 (12) (2010) i367–i373.
[22] C. Trapnell, S.L. Salzberg, How to map billions of short reads onto genomes, Nature Biotechnology 27 (5) (2009).
[23] N. Välimäki, S. Ladra, V. Mäkinen, Approximate all-pairs suffix/prefix overlaps, in: Proc. 21st Annual Symposium on Combinatorial Pattern Matching, in: Lecture Notes in Computer Science, vol. 6129, Springer-Verlag, Berlin, 2010, pp. 76–87.