Space-Efficient Construction of Compressed Suffix Trees☆

Nicola Prezza[a,b], Giovanna Rosone[b,*]

[a] *Luiss Guido Carli, Rome, Italy*
[b] *Department of Computer Science, University of Pisa, Italy*

Abstract

We show how to build several data structures of central importance to string processing by taking as input the Burrows-Wheeler transform (BWT) and using small extra working space. Let n be the text length and σ be the alphabet size. We first provide two algorithms that enumerate all LCP values and suffix tree intervals in $O(n \log \sigma)$ time using just $o(n \log \sigma)$ bits of working space on top of the input re-writable BWT. Using these algorithms as building blocks, for any parameter $0 < \epsilon \leq 1$ we show how to build the PLCP bitvector and the balanced parentheses representation of the suffix tree topology in $O\left(n(\log \sigma + \epsilon^{-1} \cdot \log \log n)\right)$ time using at most $n \log \sigma \cdot (\epsilon + o(1))$ bits of working space on top of the input re-writable BWT and the output. For example, we can build a compressed suffix tree from the BWT using just succinct working space (i.e. $o(n \log \sigma)$ bits) and $\Theta\left(n \log \sigma + n(\log \log n)^{1+\delta}\right)$ time, for any constant $\delta > 0$. This improves the previous most space-efficient algorithms, which worked in $O(n)$ bits and $O(n \log n)$ time. We also consider the problem of merging BWTs of string collections, and provide a solution running in $O(n \log \sigma)$ time and using just $o(n \log \sigma)$ bits of working space. An efficient implementation of our LCP construction and BWT merge algorithms uses (in RAM) as few as n bits on top of a packed representation of the input/output and process data as fast as 2.92 megabases per second.

Keywords: Burrows-Wheeler transform, compressed suffix tree, LCP, PLCP.

*Corresponding author

Email addresses: `nprezza@luiss.it` (Nicola Prezza), `giovanna.rosone@unipi.it` (Giovanna Rosone)

1. Introduction and Related Work

The increasingly-growing production of large string collections—especially in domains such as biology, where new generation sequencing technologies can nowadays generate gigabytes of data in few hours—is lately generating much interest towards fast and space-efficient algorithms able to index this data. The Burrows-Wheeler Transform [1] and its extension to sets of strings [2, 3] is becoming the gold-standard in the field: even when not compressed, its size is asymptotically smaller than classic suffix arrays (while preserving many of their indexing capabilities). This generated considerable interest towards fast and space-efficient BWT construction algorithms [3, 4, 5, 6, 7, 8, 9, 5]. As a result, the problem of building the BWT is well understood to date. The fastest algorithm solving this problem operates in sublinear $O(n/\sqrt{\log n})$ time and $O(n)$ bits of space on a binary text of length n by exploiting word parallelism [8]. The authors also provide a conditional lower bound suggesting that this running time might be optimal. On general alphabets, the most space-efficient algorithm terminates in $O(n \log n / \log\log n)$ time and uses just $o(n \log \sigma)$ bits of space (succinct) on top of the input and compressed output [9], where σ is the alphabet's size. In the average case, this running time can be improved to $O(n)$ on constant-sized alphabets while still operating within succinct space [5].

In some cases, a BWT alone is not sufficient to complete efficiently particular string-processing tasks. For this reason, the functionalities of the BWT are often extended by augmenting it with additional structures such as the Longest Common Prefix (LCP) array [10] (see e.g. [11, 12, 13, 14] for bioinformatic applications requiring this additional component). A disadvantage of the LCP array is that it requires $O(n \log n)$ bits to be stored in plain form. To alleviate this problem, usually the PLCP array [15]—an easier-to-compress permutation of the LCP array—is preferred. The PLCP relies on the idea of storing LCP values in text order instead of suffix array order. As shown by Kasai et al. [16], this permutation is almost increasing ($PLCP[i+1] \geq PLCP[i] - 1$) and can thus be represented in just $2n$ bits in a bitvector known as the *PLCP bitvector*. More advanced applications might even require full suffix tree functionality. In such cases, compressed suffix trees [17, 18] (CSTs) are the preferred choice when the space is at a premium. A typical compressed suffix tree is formed by a compressed suffix array (CSA), the PLCP bitvector, and a succinct representation of the suffix tree topology [18] (there exist other designs, see Ohlebusch et al. [19] for an exhaustive survey). To date, several practical algorithms have been developed to solve the task of building *de novo* such additional components [10, 20, 21, 22, 23, 24, 25, 26], but little work has been devoted to the task of computing them from the BWT in little working space (internal and external). Considering the advanced point reached by state-of-the-art BWT construction algorithms, it is worth exploring whether such structures can be built more efficiently starting from the BWT, rather than from the raw input text.

CSA As far as the CSA is concerned, this component can be easily built from the BWT using small space as it is formed (in its simplest design) by just

a BWT with rank/select functionality enhanced with a suffix array sampling, see also [24].

LCP We are aware of only one work building the LCP array in small space from the BWT: Beller et al. [27] show how to build the LCP array in $O(n \log \sigma)$ time and $O(n)$ bits of working space on top of the input BWT and the output.

PLCP Kärkkäinen et al. [25] show that the PLCP bitvector can be built in $O(n \log n)$ time using n bits of working space on top of the text, the suffix array, and the output PLCP. Kasai at al.'s lemma also stands at the basis of a more space-efficient algorithm from Välimäki et al. [26], which computes the PLCP from a CSA in $O(n \log n)$ time using constant working space on top of the CSA and the output. Belazzougui [24] recently presented an algorithm for building the PLCP bitvector from the text in $O(n)$ randomized time and compact space ($O(n \log \sigma)$ bits). This was later improved by Munro et al. [28], who made the running time deterministic.

Suffix tree topology The remaining component required to build a compressed suffix tree (in the version described by Sadakane [18]) is the suffix tree topology, represented either in BPS [29] (balanced parentheses) or DFUDS [30] (depth first unary degree sequence), using $4n$ bits. As far as the BPS representation is concerned, Hon et al. [31] show how to build it from a CSA in $O(n(\log \sigma + \log^{\epsilon} n))$ time and compact space for any constant $\epsilon > 0$. Belazzougui [24] improves this running time to the optimal $O(n)$, still working within compact space. Välimäki et al. [26] describe a linear-time algorithm that improves the space to $O(n)$ bits on top of the LCP array (which however needs to be represented in plain form), while Ohlebusch et al. [19] show how to build the DFUDS representation of the suffix tree topology in $O(t_{lcp} \cdot n)$ time using $n + o(n)$ bits of working space on top of a structure supporting access to LCP array values in $O(t_{lcp})$ time.

Summing up, the situation for building compressed suffix trees from the BWT is the following: algorithms working in optimal linear time require $O(n \log \sigma)$ bits of working space. Algorithms reducing this space to $O(n)$ (on top of a CSA) are only able to build the suffix tree topology within $O(n \cdot t_{lcp})$ time, which is $\Omega(n \log^{\epsilon} n)$ with the current best techniques, and the PLCP bitvector in $O(n \log n)$ time. No algorithm can build all the three CST components within $o(n \log \sigma)$ bits of working space on top of the input BWT and the output. Combining the most space-efficient existing algorithms, the following two trade-offs can therefore be achieved for building all compressed suffix tree components from the BWT:

- $O(n \log \sigma)$ bits of working space and $O(n)$ time, or
- $O(n)$ bits of working space and $O(n \log n)$ time.

Our contributions. In this paper, we give new space-time trade-offs that allow building the CST's components in smaller working space (and in some cases even faster) with respect to the existing solutions. We start by combining Beller et al.'s algorithm [27] with the suffix-tree enumeration procedure of Belazzougui [24] to obtain an algorithm that enumerates (i) all pairs $(i, LCP[i])$,

and (ii) all suffix tree intervals in $O(n \log \sigma)$ time using just $o(n \log \sigma)$ bits of working space on top of the input re-writable BWT. We use this procedure to obtain algorithms that build (working space is on top of the input BWT and the output):

1. The LCP array of a string collection in $O(n \log \sigma)$ time and $o(n \log \sigma)$ bits of working space (Section 5).
2. the PLCP bitvector and the BPS representation of the suffix tree topology in $O\left(n(\log \sigma + \epsilon^{-1} \cdot \log \log n)\right)$ time and $n \log \sigma \cdot (\epsilon + o(1))$ bits of working space, for any user-defined parameter $0 < \epsilon \leq 1$ (Section 7 and 8).
3. The BWT of the union of two string collections of total size n in $O(n \log \sigma)$ time and $o(n \log \sigma)$ bits of working space, given the BWTs of the two collections as input (Section 9).

Contribution (1) is the first showing that the LCP array can be induced from the BWT using succinct working space *for any alphabet size*.

Contribution (2) can be used to build a compressed suffix tree from the BWT using just $o(n \log \sigma)$ bits of working space and $\Theta\left(n \log \sigma + n(\log \log n)^{1+\delta}\right)$ time, for any constant $\delta > 0$. On small alphabets, this improves both working space and running time of existing $O(n)$-bits solutions.

Also contribution (3) improves the state-of-the-art, due to Belazzougui et al. [24, 32]. In those papers, the authors show how to merge the BWTs of two texts T_1, T_2 and obtain the BWT of the collection $\{T_1, T_2\}$ in $O(nk)$ time and $n \log \sigma(1 + 1/k) + 11n + o(n)$ bits of working space for any $k \geq 1$ [32, Thm. 7]. When $k = \log \sigma$, this running time is the same as our result (3), but the working space is much higher on small alphabets.

We implemented and tested our algorithms (1) and (3) on DNA alphabet. Our tools use (in RAM) as few as n bits on top of a packed representation of the input/output, and process data as fast as 2.92 megabases per second.

Contributions (1) and (3) are part of a preliminary version [33] of this paper. This paper also extends such results with the suffix tree interval enumeration procedure and with the algorithms of contribution (2) for building the PLCP bitvector and the BPS representation of the suffix tree topology.

2. Basic Concepts

Let $\Sigma = \{c_1, c_2, \ldots, c_\sigma\}$ be a finite ordered alphabet of size σ with $\# = c_1 < c_2 < \ldots < c_\sigma$, where $<$ denotes the standard lexicographic order. Given a text $T = t_1 t_2 \cdots t_n \in \Sigma^*$ we denote by $|T|$ its length n. We assume that the input text is terminated by the special symbol (terminator) $\#$, which does not appear elsewhere in T. We use ϵ to denote the empty string. A *factor* (or *substring*) of T is written as $T[i, j] = t_i \cdots t_j$ with $1 \leq i \leq j \leq n$. When declaring an array A, we use the same notation $A[1, n]$ to indicate that the array has n entries indexed from 1 to n. A *right-maximal* substring W of T is a string for which there exist at least two distinct characters a, b such that Wa and Wb occur in T.

The *suffix array* SA of a string T (see [34] for a survey) is an array containing the permutation of the integers $1, 2, \ldots, n$ that arranges the starting positions of the suffixes of T into lexicographical order, i.e., for all $1 \leq i < j \leq n$, $T[SA[i], n] < T[SA[j], n]$.

The *inverse suffix array* $ISA[1, n]$ is the inverse permutation of SA, i.e., $ISA[i] = j$ if and only if $SA[j] = i$.

The Burrows-Wheeler Transform of a string T is a reversible transformation that permutes its symbols, i.e. $BWT[i] = T[SA[i] - 1]$ if $SA[i] > 1$ or # otherwise.

In some of our results we deal with *string collections*. There exist some natural extensions of the suffix array and the Burrows-Wheeler Transform to a collection of strings.

Let $\mathcal{S} = \{T_1, \ldots, T_m\}$ be a string collection of total length n, where each T_i is terminated by a character # (the terminator) lexicographically smaller than all other alphabet's characters. In particular, a collection is an ordered multiset, and we denote $\mathcal{S}[i] = T_i$.

We define lexicographic order among the strings' suffixes in the usual way, except that, *only while sorting*, each terminator # of the i-th string $\mathcal{S}[i]$ is considered (implicitly) a different symbol $\#_i$, with $\#_i < \#_j$ if and only if $i < j$. Equivalently, in case of equal suffixes ties are broken by input's order: if $T_i[k, |T_i| - 1] = T_j[k', |T_j| - 1]$, then we define $T_i[k, |T_i|] < T_j[k', |T_j|]$ if and only if $i < j$.

The *generalized suffix array* $GSA[1, n]$ (see [35, 10, 36]) of $\mathcal{S}$ is an array of pairs $GSA[i] = \langle j, k \rangle$ such that $\mathcal{S}[j][k, |\mathcal{S}[j]|]$ is the i-th lexicographically smallest suffix of strings in $\mathcal{S}$, where we break ties by input position (i.e. j in the notation above). Note that, if the collection is formed by a single string T, then the first component in GSA's pairs is always equal to 1, and the second components form the suffix array of T. We denote by $\texttt{range(W)} = \langle \texttt{left(W)}, \texttt{right(W)} \rangle$, also referred to as *suffix array (SA) interval of W, or simply W-interval*, the maximal pair $\langle L, R \rangle$ such that all suffixes in $GSA[L, R]$ are prefixed by W. We use the same notation with the suffix array of a single string T. Note that the number of suffixes lexicographically smaller than W in the collection is $L - 1$. We extend this definition also to cases where W is not present in the collection: in this case, the (empty) range is $\langle L, L - 1 \rangle$ and we still require that $L - 1$ is the number of suffixes lexicographically smaller than W in the collection (or in the string).

The *extended Burrows-Wheeler Transform* $BWT[1, n]$ [2, 3] of $\mathcal{S}$ is the character array defined as $BWT[i] = \mathcal{S}[j][k - 1 \bmod |\mathcal{S}[j]|]$, where $\langle j, k \rangle = GSA[i]$.

To simplify notation, we indicate with "BWT" both the Burrows-Wheeler Transform of a string and of a string collection. The used transform will be clear from the context.

The *longest common prefix* (LCP) array of a string s [37] (resp. a collection $\mathcal{S}$ of strings, see [10, 36, 23]) is an array storing the lengths of the longest common prefixes between two consecutive suffixes of s (resp. $\mathcal{S}$) in lexicographic order (with $LCP[1] = 0$). More formally, $LCP[1] = 0$ and $LCP[i]$, for $i > 1$, stores the length of the longest common prefix between the suffixes $T[SA[i], n]$ and

$T[SA[i-1], n]$ on a text T, or the length of the longest common prefix between the substrings $\mathcal{S}[j][k, |\mathcal{S}[j]| - 1]$ and $\mathcal{S}[j'][k', |\mathcal{S}[j']| - 1]$, where $\langle j, k\rangle = GSA[i]$ and $\langle j', k'\rangle = GSA[i-1]$, respectively, on a string collection $\mathcal{S}$ (note that, by this definition, the strings' terminators do not contribute to $LCP[i]$).

Given two collections $\mathcal{S}_1, \mathcal{S}_2$ of total length n, the Document Array of their union is the binary array $DA[1, n]$ such that $DA[i] = 0$ if and only if the i-th smallest suffix comes from $\mathcal{S}_1$. When merging suffixes of the two collections, ties are broken by collection number (i.e. suffixes of $\mathcal{S}_1$ are smaller than suffixes of $\mathcal{S}_2$ in case of ties).

The C-array of a string (or collection) S is an array $C[1, \sigma]$ such that $C[i]$ contains the number of characters lexicographically smaller than i in S, plus one (S will be clear from the context). Equivalently, $C[c]$ is the starting position of suffixes starting with c in the suffix array of the string. When S (or any of its permutations) is represented with a balanced wavelet tree, then we do not need to store explicitly C, and $C[c]$ can be computed in $O(\log \sigma)$ time with no space overhead on top of the wavelet tree (see [38]). Function $\texttt{S.rank}_\texttt{c}(\texttt{i})$ returns the number of characters equal to c in $S[1, i-1]$. When S is represented by a wavelet tree, *rank* can be computed in $O(\log \sigma)$ time.

Function $\texttt{getIntervals}(\texttt{L}, \texttt{R}, \texttt{BWT})$, where BWT is the extended Burrows-Wheeler transform of a string collection $\mathcal{S}$ and $\langle L, R\rangle$ is the suffix array interval of some string W appearing as a substring of some element of $\mathcal{S}$, returns all suffix array intervals of strings cW, with $c \neq \#$, that occur in $\mathcal{S}$. When BWT is represented with a balanced wavelet tree, we can implement this function so that it terminates in $O(\log \sigma)$ time per returned interval [27]. The function can be made to return the output intervals on-the-fly, one by one (in an arbitrary order), without the need to store them all in an auxiliary vector, with just $O(\log n)$ bits of additional overhead in space [27] (this requires a DFS-visit of the wavelet tree's subtree induced by $BWT[L, R]$; the visit requires only $\log \sigma$ bits to store the current path in the tree).

An extension of the above function that navigates in parallel two BWTs is immediate. Function $\texttt{getIntervals}(\texttt{L}_1, \texttt{R}_1, \texttt{L}_2, \texttt{R}_2, \texttt{BWT}_1, \texttt{BWT}_2)$ takes as input two ranges of a string W on the BWTs of two collections, and returns the pairs of ranges on the two BWTs corresponding to all left-extensions cW of W ($c \neq \#$) such that cW appears in at least one of the two collections. To implement this function, it is sufficient to navigate in parallel the two wavelet trees as long as at least one of the two intervals is not empty.

Let S be a string. The function $S.\texttt{rangeDistinct}(\texttt{i}, \texttt{j})$ returns the set of distinct alphabet characters *different than the terminator* # in $S[i, j]$. Also this function can be implemented in $O(\log \sigma)$ time per returned element when S is represented with a wavelet tree (again, this requires a DFS-visit of the sub-tree of the wavelet tree induced by $S[i, j]$).

$BWT.\texttt{bwsearch}(\langle \texttt{L}, \texttt{R}\rangle, \texttt{c})$ is the function that, given the suffix array interval $\langle L, R\rangle$ of a string W occurring in the collection, returns the suffix array interval of cW by using the BWT of the collection [39]. This function requires access to array C and *rank* support on BWT, and runs in $O(\log \sigma)$ time when BWT is represented with a balanced wavelet tree.

To conclude, our algorithms will take as input a wavelet tree representing the BWT. As shown in the next lemma by Claude et al., this is not a restriction:

Lemma 1 ([40])**.** *Given a re-writable word-packed string of length n on alphabet $[1, \sigma]$, we can replace it with its wavelet matrix [40] in $O(n \log \sigma)$ time using n bits of additional working space.*

Wavelet matrices [40] are a space-efficient representation of wavelet trees taking $n \log \sigma \cdot (1 + o(1))$ bits of space and supporting all their operations within the same running times. Since the output of all our algorithms will be at least n bits, it will always be possible to re-use a portion of the output's space (before computing it) to fit the extra n bits required by Lemma 1.

3. Belazzougui's Enumeration Algorithm

In [24], Belazzougui showed that a BWT with *rank* and *range distinct* functionality (see Section 2) is sufficient to enumerate in small space a rich representation of the internal nodes of the suffix tree of a text T. For the purposes of this article, we assume that the BWT is represented using a wavelet tree (whereas Belazzougui's original result is more general), and thus that all queries take $O(\log \sigma)$ time.

Theorem 1 (Belazzougui [24])**.** *Given the Burrows-Wheeler Transform of a text $T \in [1, \sigma]^n$ represented with a wavelet tree, we can enumerate the following information for each distinct right-maximal substring W of T: (i) $|W|$, and (ii) range(Wc_i) for all $c_1 < \cdots < c_k$ such that Wc_i occurs in T. The process runs in $O(n \log \sigma)$ time and uses $O(\sigma^2 \log^2 n)$ bits of working space on top of the BWT.*

To keep the article self-contained, in this section we describe the algorithm at the core of the above result. Remember that explicit suffix tree nodes correspond to right-maximal substrings. The first idea is to represent any substring W (not necessarily right-maximal) as follows. Let $\texttt{chars}_\texttt{W}[\texttt{1}, \texttt{k}_\texttt{W}]$ be the alphabetically-sorted character array such that $W \cdot \texttt{chars}_\texttt{W}[\texttt{i}]$ is a substring of T for all $i = 1, \ldots, k_W$, where k_W is the number of right-extensions of W. We require $\texttt{chars}_\texttt{W}$ to be also complete: if Wc is a substring of T, then $c \in \texttt{chars}_\texttt{W}$. Let moreover $\texttt{first}_\texttt{W}[\texttt{1}, \texttt{k}_\texttt{W} + \texttt{1}]$ be the array such that $\texttt{first}_\texttt{W}[\texttt{i}]$ is the starting position of (the range of) $W \cdot \texttt{chars}_\texttt{W}[\texttt{i}]$ in the suffix array of T for $i = 1, \ldots, k_W$, and $\texttt{first}_\texttt{W}[\texttt{k}_\texttt{W} + \texttt{1}]$ is the end position of W in the suffix array of T. The representation for W is (differently from [24], we omit $\texttt{chars}_\texttt{W}$ from the representation and we add $|W|$; these modifications will turn out to be useful later):

$$\texttt{repr(W)} = \langle \texttt{first}_\texttt{W},\ |\texttt{W}| \rangle$$

Note that, if W is not right-maximal nor a text suffix, then W is followed by $k_W = 1$ distinct characters in T and the above representation is still well-defined.

When W is right-maximal, we will also say that $\texttt{repr}(\texttt{W})$ is the representation of a suffix tree explicit node (i.e. the node reached by following the path labeled W from the root).

Weiner Link Tree Visit. The enumeration algorithm works by visiting the Weiner Link tree of T starting from the root's representation, that is, $\texttt{repr}(\epsilon) = \langle \texttt{first}_\epsilon,\ 0 \rangle$, where $\texttt{first}_\epsilon = \langle C[c_1], \ldots, C[c_\sigma], n \rangle$ (see Section 2 for a definition of the C-array) and $c_1, \ldots, c_\sigma$ are the sorted alphabet's characters. Since the suffix tree and the Weiner link tree share the same set of nodes, this is sufficient to enumerate all suffix tree nodes. The visit uses a stack storing representations of suffix tree nodes, initialized with $\texttt{repr}(\epsilon)$. At each iteration, we pop the head $\texttt{repr}(\texttt{W})$ from the stack and we push $\texttt{repr}(\texttt{cW})$ such that cW is right-maximal in T. To keep the stack's size under control, once we have computed $\texttt{repr}(\texttt{cW})$ for the right-maximal left-extensions cW of W we push them on the stack in decreasing order of range length $\texttt{range}(\texttt{cW})$ (i.e. the node with the smallest range is pushed last). This guarantees that the stack will always contain at most $O(\sigma \log n)$ elements [24]. Since each element takes $O(\sigma \log n)$ bits to be represented, the stack's size never exceeds $O(\sigma^2 \log^2 n)$ bits.

Computing Weiner Links. We now show how to efficiently compute the node representation $\texttt{repr}(\texttt{cW})$ from $\texttt{repr}(\texttt{W})$ for the characters c such that cW is right-maximal in T. In [24, 32] this operation is supported efficiently by first enumerating all *distinct* characters in each range $BWT[\texttt{first}_\texttt{W}[\texttt{i}], \texttt{first}_\texttt{W}[\texttt{i}+1]]$ for $i = 1, \ldots, k_W$, using function $\texttt{BWT.rangeDistinct}(\texttt{first}_\texttt{W}[\texttt{i}], \texttt{first}_\texttt{W}[\texttt{i}+1])$ (see Section 2). Equivalently, for each $a \in \texttt{chars}_\texttt{W}$ we want to list all distinct left-extensions cWa of Wa. Note that, in this way, we may also visit implicit suffix tree nodes (i.e. some of these left-extensions could be not right-maximal). Stated otherwise, we are traversing all explicit *and* implicit Weiner links. Since the number of such links is linear [24, 41] (even including implicit Weiner links[1]), globally the number of distinct characters returned by `rangeDistinct` operations is $O(n)$. An implementation of `rangeDistinct` on wavelet trees is discussed in [27] with the procedure `getIntervals` (this procedure actually returns more information: the suffix array range of each cWa). This implementation runs in $O(\log \sigma)$ time per returned character. Globally, we therefore spend $O(n \log \sigma)$ time using a wavelet tree. We now need to compute $\texttt{repr}(\texttt{cW})$ for all left-extensions of W and keep only the right-maximal ones. Let $x = \texttt{repr}(\texttt{W})$ and $\texttt{BWT.Weiner}(\texttt{x})$ be the function that returns the representations of such strings (used in Line 12 of Algorithm 1). This function can be

[1] To see this, first note that the number of right-extensions Wa of W that have only one left-extension cWa is at most equal to the number of right-extensions of W; globally, this is at most the number of suffix tree's nodes (linear). Any other right-extension Wa that has at least two distinct left-extensions cWa and bWa is, by definition, left maximal and corresponds therefore to a node in the suffix tree of the reverse of T. It follows that all left-extensions of Wa can be charged to an edge of the suffix tree of the reverse of T (again, the number of such edges is linear).

implemented by observing that

$$\texttt{range(cWa)} = \langle \begin{array}{l} \texttt{C[c]} + \texttt{BWT.rank}_\texttt{c}(\texttt{left(Wa)}), \\ \texttt{C[c]} + \texttt{BWT.rank}_\texttt{c}(\texttt{right(Wa)} + 1) - 1 \rangle \end{array}$$

where $a = \texttt{chars}_\texttt{W}[\texttt{i}]$ for $1 \leq i < |\texttt{first}_\texttt{W}|$, and noting that $\texttt{left(Wa)}$ and $\texttt{right(Wa)}$ are available in $\texttt{repr(W)}$. Note also that we do not actually need to know the value of characters $\texttt{chars}_\texttt{W}[\texttt{i}]$ to compute the ranges of each $cW \cdot \texttt{chars}_\texttt{W}[\texttt{i}]$; this is the reason why we can omit $\texttt{chars}_\texttt{W}$ from $\texttt{repr(W)}$. Using a wavelet tree, the above operation takes $O(\log \sigma)$ time. By the above observations, the number of strings cWa such that W is right-maximal is bounded by $O(n)$. Overall, computing $\texttt{repr(cW)} = \langle \texttt{first}_\texttt{cW}, |W| + 1 \rangle$ for all left-extensions cW of all right-maximal strings W takes therefore $O(n \log \sigma)$ time. Within the same running time, we can check which of those extensions is right maximal (i.e. those such that $|\texttt{first}_\texttt{cW}| \geq 2$), sort them in-place by interval length (we always sort at most σ node representations, therefore also sorting takes globally $O(n \log \sigma)$ time), and push them on the stack.

4. Beller et al.'s Algorithm

The second ingredient used in our solutions is the following result, due to Beller et al. (we slightly re-formulate their result to fit our purposes, read below for a description of the differences):

Theorem 2 (Beller et al.[27])**.** *Given the Burrows-Wheeler Transform of a text T represented with a wavelet tree, we can enumerate all pairs $(i, LCP[i])$ in $O(n \log \sigma)$ time using $5n$ bits of working space on top of the BWT.*

Theorem 2 represents the state of the art for computing the LCP array from the BWT. Also Beller et al.'s algorithm works by enumerating a (linear) subset of the BWT intervals. LCP values are induced from a particular visit of those intervals. Belazzougui's and Beller et al.'s algorithms have, however, two key differences which make the former more space-efficient on small alphabets, while the latter is more space-efficient on large alphabets: (i) Beller et al. use a queue (FIFO) instead of a stack (LIFO), and (ii) they represent W-intervals with just the pair of coordinates $\texttt{range(W)}$ and the value $|W|$. In short, while Beller et al.'s queue might grow up to size $\Theta(n)$, the use of intervals (instead of the more complex representation used by Belazzougui) makes it possible to represent it using $O(1)$ bitvectors of length n. On the other hand, the number of items on Belazzougui's stack can be upper-bounded by $O(\sigma \log n)$, but its elements take more space to be represented.

We now describe in detail Beller et al.'s result. We keep a bitvector $U[1, n]$ such that $U[i] = 0$ if and only if the pair $(i, LCP[i])$ has not been output yet. In their original algorithm, Beller et al. use the LCP array itself to mark undefined LCP entries. In our case, we do not want to store the whole LCP array (for reasons that will be clear in the next sections) and thus we only

record which LCP values have been output. Bitvector U accounts for the additional n bits used by Theorem 2 with respect to the original result described in [27]. At the beginning, $U[i] = 0$ for all $i = 1, \ldots, n$. Beller et al.'s algorithm starts by inserting in the queue the triple $\langle 1, n, 0 \rangle$, where the first two components are the BWT interval of ϵ (the empty string) and the third component is its length. From this point, the algorithm keeps performing the following operations until the queue is empty. We remove the first (i.e. the oldest) element $\langle L, R, \ell \rangle$ from the queue, which (by induction) is the interval and length of some string W: $\texttt{range}(\texttt{W}) = \langle L, R \rangle$ and $|W| = \ell$. Using operation $\texttt{getIntervals}(\texttt{L}, \texttt{R}, \texttt{BWT})$ [27] (see Section 2) we left-extend the BWT interval $\langle L, R \rangle$ with the characters $c_1, \ldots, c_k$ in $\texttt{BWT.rangeDistinct}(\texttt{L}, \texttt{R})$, obtaining the triples $\langle L_1, R_1, \ell+1 \rangle, \ldots, \langle L_k, R_k, \ell+1 \rangle$ corresponding to the strings $c_1 W, \ldots, c_k W$. For each such triple $\langle L_i, R_i, \ell + 1 \rangle$, if $R_i \neq n$ and $U[R_i + 1] = 0$ then we set $U[R_i + 1] \leftarrow 1$, we output the LCP pair $(R_i + 1, \ell)$ and push $\langle L_i, R_i, \ell + 1 \rangle$ on the queue. Importantly, note that we can push the intervals returned by $\texttt{getIntervals}(\texttt{L}, \texttt{R}, \texttt{BWT})$ in the queue in any order; as discussed in Section 2, this step can be implemented with just $O(\log n)$ bits of space overhead with a DFS-visit of the wavelet tree's sub-tree induced by $BWT[L, R]$ (i.e. the intervals are not stored temporarily anywhere: they are pushed as soon as they are generated).

Queue implementation. To limit space usage, Beller et al. use the following queue representations. First note that, at each time point, the queue's triples are partitioned into a (possibly empty) sequence with associated length (i.e. the third element in the triples) $\ell + 1$, followed by a sequence with associated length ℓ, for some ℓ. To simplify the description, let us assume that these two sequences are kept as two distinct queues, indicated in the following as Q_ℓ and $Q_{\ell+1}$. At any stage of the algorithm, we pop from Q_ℓ and push into $Q_{\ell+1}$. It follows that there is no need to store strings' lengths in the triples themselves (i.e. the queue's elements become just ranges), since the length of each element in Q_ℓ is ℓ. When Q_ℓ is empty, we create a new empty queue $Q_{\ell+2}$, pop from $Q_{\ell+1}$, and push into $Q_{\ell+2}$ (and so on). Beller et al. represent Q_ℓ as follows. While pushing elements in Q_ℓ, as long as its size does not exceed $n/\log n$ we represent it as a vector of pairs (of total size at most $O(n)$ bits). This representation supports push/pop operations in (amortized) constant time and takes at most $O(\log n \cdot n/\log n) = O(n)$ bits of space. As soon as Q_ℓ's size exceeds $n/\log n$, we switch to a representation that uses two packed bitvectors $\texttt{open}[\texttt{1}, \texttt{n}]$ and $\texttt{close}[\texttt{1}, \texttt{n}]$ storing, respectively, the left- and right-most boundaries of the ranges in the queue. Note that this representation can be safely used since the pairs in Q_ℓ are suffix array ranges of strings of some fixed length ℓ, therefore there cannot be overlapping intervals. Pushing an interval into such a queue takes constant time (it just requires setting two bits). Popping all the $t = |Q_\ell|$ intervals, on the other hand, can easily be implemented in $O(t + n/\log n)$ time by scanning the bitvectors and exploiting word-parallelism: since $\texttt{open}[\texttt{1}, \texttt{n}]$ is packed into $n/\log n$ words of $\log n$ bits each, it is sufficient to scan it left-to-right in $O(n/\log n)$ time in order to locate words containing at least one bit set.

Then, the position of the leftmost bit set in the word can be found in $O(1)$ time by using a standard universal table of size $O(\sqrt{n}\log n)$ bits indexing all combinations of $\log n/2$ bits. Since Beller et al.'s procedure visits $O(n)$ SA intervals, Q_ℓ will exceed size $n/\log n$ for at most $O(\log n)$ values of ℓ. It follows that also with this queue representation pop operations take amortized constant time.

Time complexity. It is easy to see that the algorithm inserts in total a linear number of intervals in the queue since an interval $\langle L_i, R_i, \ell+1\rangle$ is inserted only if $U[R_i+1]=0$, and successively $U[R_i+1]$ is set to 1. Clearly, this can happen at most n times. In [27] the authors moreover show that, even when counting the left-extensions of those intervals (computed after popping each interval from the queue), the total number of generated intervals stays linear. Overall, the algorithm runs therefore in $O(n\log\sigma)$ time (as discussed in Section 2, `getIntervals` runs in $O(\log\sigma)$ time per returned element).

5. Enumerating LCP values

In this section we prove our first main result: how to enumerate LCP pairs $(i, LCP[i])$ using succinct working space on top of a wavelet tree representing the BWT. Later we will use this procedure to build the LCP and PLCP arrays in small space on top of a plain representation of the BWT. We give our lemma in the general form of string collections, which will require adapting the algorithms seen in the previous sections to this more general setting. Our first observation is that Theorem 1, extended to string collections as described below, can be directly used to enumerate LCP pairs $(i, LCP[i])$ using just $O(\sigma^2\log^2 n)$ bits of working space on top of the input and output. We combine this procedure with an extended version of Beller et al.'s algorithm working on string collections in order to get small working space for all alphabets. Algorithms 1 and 2 report our complete procedure; read below for an exhaustive description. We obtain our first main result:

Lemma 2. *Given a wavelet tree for the Burrows-Wheeler Transform of a collection $\mathcal{S}=\{T_1,\ldots,T_m\}$ of total length n on alphabet $[1,\sigma]$, we can enumerate all pairs $(i, LCP[i])$ in $O(n\log\sigma)$ time using $o(n\log\sigma)$ bits of working space on top of the BWT.*

Proof. If $\sigma < \sqrt{n}/\log^2 n$ then $\sigma^2\log^2 n = o(n)$ and our extension of Theorem 1 gives us $o(n\log\sigma)$ additional working space. If $\sigma \geq \sqrt{n}/\log^2 n$ then $\log\sigma = \Theta(\log n)$ and we can use our extension to string collections of Theorem 2, which yields extra working space $O(n) = o(n\log n) = o(n\log\sigma)$. Note that, while we used the threshold $\sigma < \sqrt{n}/\log^2 n$, any threshold of the form $\sigma < \sqrt{n}/\log^{1+\epsilon} n$, with $\epsilon > 0$ would work. The only constraint is that $\epsilon > 0$, since otherwise for $\epsilon = 0$ the working space would become $O(n\log\sigma)$ for constant σ (which is no good because we aim at $o(n\log\sigma)$). □

We now describe all the details of our extensions of Theorems 1 and 2 used in the proof of Lemma 2. Procedure `BGOS(BWT)` in Line 2 of Algorithm 1 is a call to Beller et al.'s algorithm, modified as follows. First, we enumerate the LCP pairs $(C[c], 0)$ for all $c \in \Sigma$. Then, we push in the queue $\langle \texttt{range(c)}, \texttt{1} \rangle$ for all $c \in \Sigma$ and start the main algorithm. Note moreover that (see Section 2) from now on we never left-extend ranges with #.

Recall that each string of a text collection $\mathcal{S}$ is ended by a terminator # that is equal for all strings. Consider now the LCP and GSA arrays of $\mathcal{S}$. We divide LCP values into two types. A LCP value `LCP[i]`, with $i > 1$, is of *node type* when the i-th and $(i-1)$-th suffixes (which includes the two equal terminators) are distinct: $\mathcal{S}[j][k, |\mathcal{S}[j]|] \neq \mathcal{S}[j'][k', |\mathcal{S}[j']|]$, where $GSA[i] = \langle j, k \rangle$ and $GSA[i-1] = \langle j', k' \rangle$. Those two suffixes differ before the terminator is reached in both suffixes (it might be reached in one of the two suffixes, however); we use the name *node-type* because $i-1$ and i are the last and first suffix array positions of the ranges of two adjacent children of some suffix tree node, respectively (i.e. the node corresponding to string $\mathcal{S}[j][k, k+LCP[i]-1]$). Note that it might be that one of the two suffixes, $\mathcal{S}[j][k, |\mathcal{S}[j]|]$ or $\mathcal{S}[j'][k', |\mathcal{S}[j']|]$, is the string "#". Similarly, a *leaf-type* LCP value `LCP[i]`, with $i > 1$, is such that the i-th and $(i-1)$-th suffixes are equal: $\mathcal{S}[j][k, |\mathcal{S}[j]|] = \mathcal{S}[j'][k', |\mathcal{S}[j']|]$. We use the name *leaf-type* because, in this case, it must be the case that $i \in [L+1, R]$, where $\langle L, R \rangle$ is the suffix array range of some suffix tree leaf (it might be that $R > L$ since there might be repeated suffixes in the collection). Note that, in this case, $\mathcal{S}[j][k, |\mathcal{S}[j]|] = \mathcal{S}[j'][k', |\mathcal{S}[j']|]$ could coincide with #. Entry `LCP[1]` escapes the above classification, so we output it separately.

Our idea is to compute first node-type and then leaf-type LCP values. We argue that Beller et al.'s algorithm already computes the former kind of LCP values. When this algorithm uses too much space (i.e. on small alphabets), we show that Belazzougui's enumeration strategy can be adapted to reach the same goal: by the very definition of node-type LCP values, they lie between children of some suffix tree node x, and their value corresponds to the string depth of x. This strategy is described in Algorithm 1. Function `BWT.Weiner(x)` in Line 12 takes as input the representation of a suffix tree node x and returns all explicit nodes reached by following Weiner links from x (an implementation of this function is described in Section 3). Figure 1 shows how node-type LCP values are computed using Algorithm 1. Leaf-type LCP values, on the other hand, can easily be computed by enumerating intervals corresponding to suffix tree leaves. To reach this goal, it is sufficient to enumerate ranges of suffix tree leaves starting from `range`(#) and recursively left-extending with backward search with characters different from # whenever possible. For each range $\langle L, R \rangle$ obtained in this way, we set each entry $LCP[L+1, R]$ to the string depth (terminator excluded) of the corresponding leaf. This strategy is described in Algorithm 2, and Figure 2 shows an example. In the figure, we denote with `leaf_repr` the representation we use for the leaves (that is, range of the leaf and its string depth). In order to limit space usage, we use again a stack or a queue to store leaves and their string depth (note that each leaf takes $O(\log n)$ bits to be represented): we use a queue when $\sigma > n/\log^3 n$, and a stack otherwise.

The queue is the same used by Beller et al.[27] and described in Section 4. This guarantees that the bit-size of the queue/stack never exceeds $o(n \log \sigma)$ bits: since leaves take just $O(\log n)$ bits to be represented and the stack's size never contains more than $O(\sigma \cdot \log n)$ leaves, the stack's bit-size never exceeds $O(n/\log n) = o(n)$ when $\sigma \leq n/\log^3 n$. Similarly, Beller et al's queue always takes at most $O(n)$ bits of space, which is $o(n \log \sigma)$ for $\sigma > n/\log^3 n$. Note that in Lines 18-21 we can afford storing temporarily the k resulting intervals since, in this case, the alphabet's size is small enough.

To sum up, our full procedure works as follows: (1) we output node-type LCP values using procedure `Node-Type(BWT)` described in Algorithm 1, and (2) we output leaf-type LCP values using procedure `Leaf-Type(BWT)` described in Algorithm 2.

Algorithm 1 `Node-Type(BWT)`

```
1:  if σ > √n / log² n then
2:      BGOS(BWT)                                   ▷ Run Beller et al.'s algorithm
3:  else
4:      P ← new_stack()                             ▷ Initialize new stack
5:      P.push(repr(ϵ))                             ▷ Push representation of ϵ
6:      while not P.empty() do
7:          ⟨first_W, ℓ⟩ ← P.pop()                  ▷ Pop highest-priority element
8:          t ← |first_W| − 1                       ▷ Number of children of ST node
9:          for i = 2, ..., t do
10:             output (first_W[i], ℓ)              ▷ Output LCP value
11:         end for
12:         x_1, ..., x_k ← BWT.Weiner(⟨first_W, ℓ⟩)   ▷ Follow Weiner Links
13:         x'_1, ..., x'_k ← sort(x_1, ..., x_k)   ▷ Sort by interval length
14:         for i = k ... 1 do
15:             P.push(x'_i)                        ▷ Push representations
16:         end for
17:     end while
18: end if
```

The correctness, completeness, and complexity of our procedure are proved in the following Lemma:

Lemma 3. *Algorithms 1 and 2 correctly output all LCP pairs $(i, LCP[i])$ of the collection in $O(n \log \sigma)$ time using $o(n \log \sigma)$ bits of working space on top of the input BWT.*

Proof. Correctness - Algorithm 1. We start by proving that Beller et al.'s procedure in Line 2 of Algorithm 1 (procedure `BGOS(BWT)`) outputs all the node-type LCP entries correctly. The proof proceeds by induction on the LCP value ℓ and follows the original proof of [27]. At the beginning, we insert in the queue all c-intervals, for $c \in \Sigma$. For each such interval $\langle L, R \rangle$ we output $LCP[R+1] = \ell = 0$.

Algorithm 2 `Leaf-Type(BWT)`

for $i = left(\#), \ldots, right(\#)$ **do**
  output $(i, 0)$
end for
if $\sigma > n/\log^3 n$ **then**
  $\texttt{P} \leftarrow \texttt{new\_queue}()$ ▷ Initialize new queue
else
  $\texttt{P} \leftarrow \texttt{new\_stack}()$ ▷ Initialize new stack
end if
$\texttt{P.push}(\texttt{BWT.range}(\#), 0)$ ▷ Push range of terminator and LCP value 0
while not $\texttt{P.empty}()$ **do**
  $\langle\langle L, R\rangle, \ell\rangle \leftarrow \texttt{P.pop}()$ ▷ Pop highest-priority element
  for $i = L + 1 \ldots R$ **do**
    output (i, ℓ) ▷ Output LCP inside range of ST leaf
  end for
  if $\sigma > n/\log^3 n$ **then**
    $\texttt{P.push}(\texttt{getIntervals}(\texttt{L}, \texttt{R}, \texttt{BWT}), \ell + 1)$ ▷ Pairs $\langle$interval,$\ell + 1\rangle$
  else
    $\langle L_i, R_i\rangle_{i=1,\ldots,k} \leftarrow \texttt{getIntervals}(\texttt{L}, \texttt{R}, \texttt{BWT})$
    $\langle L'_i, R'_i\rangle_{i=1,\ldots,k} \leftarrow \texttt{sort}(\langle L_i, R_i\rangle_{i=1,\ldots,k})$ ▷ Sort by interval length
    for $i = k \ldots 1$ **do**
      $\texttt{P.push}(\langle L'_i, R'_i\rangle, \ell + 1)$ ▷ Push in order of decreasing length
    end for
  end if
end while

index	LCP	BWT	Suffixes
1	0	T	#
2	0	T	#
3	0	A	#
4	0	**T**	**A** #
5	**1**	#	**A** A G C T #
6	**1**	A	**A** G C T #
7	**1**	**T**	**A** T #
8	2	**T**	**A** T A #
9	3	G	**A** T A T #
10	0	G	C T #
11	2	#	C T A T A #
12	0	#	G A T A T #
13	1	A	G C T #
14	0	C	T #
15	1	A	T #
16	1	A	**T** A #
17	**2**	A	**T** A T #
18	3	C	**T** A T A #

$\texttt{repr(A)} = \langle\langle 4, 5, 6, 7, 9\rangle, \mathbf{1}\rangle$

$\Downarrow$ `Weiner(T)`

$\texttt{repr(TA)} = \langle\langle 16, 17, 18\rangle, \mathbf{2}\rangle$

Figure 1: Running example for Algorithm 1, finding node-type LCP values with our extension of Belazzougui's strategy. **Left**. The BWT matrix and LCP array of the string collection $\{AAGCT\#, CTATA\#, GATAT\#\}$. **Top right**. Suppose we pop from the stack the representation of the right-maximal string A, of length 1. The range of A spans rows 4-9 (in orange) and the corresponding suffix tree node has four children labeled #, A, G, and T and starting at positions 4,5,6, and 7, respectively. Then, by Algorithm 1 the LCP value of positions 5,6,7 (that is, the ones corresponding to the beginning of the sub-ranges os A's children, excluded the lexicographically-smallest) is $1 = |A|$ (in blue). We therefore output the LCP pairs $\langle 5, 1\rangle$, $\langle 6, 1\rangle$, and $\langle 7, 1\rangle$. **Bottom right**. After left-extending A with T by following the corresponding Weiner link, we obtain another right-maximal string, TA, of length 2. Note that this step is performed by applying the LF mapping to the Ts in the BWT range 4-9 (in red). The ranges of the two children of suffix tree node TA begin in positions 16 (symbol #) and 17 (symbol T), therefore by Algorithm 1 the LCP value of position 17 is $2 = |TA|$ (in green) and we output the LCP pair $\langle 17, 2\rangle$.

index	LCP	BWT	Suffixes
1	0	**T**	#
2	**0**	**T**	#
3	**0**	A	#
4	0	T	A #
5	1	#	A A G C T #
6	1	A	A G C T #
7	1	T	A T #
8	2	T	A T A #
9	3	G	A T A T #
10	0	G	C T #
11	2	#	C T A T A #
12	0	#	G A T A T #
13	1	A	G C T #
14	0	C	**T** #
15	**1**	A	**T** #
16	1	A	T A #
17	2	A	T A T #
18	3	C	T A T A #

$\texttt{leaf\_repr(\#)} = \langle\langle 1, 3\rangle, \mathbf{0}\rangle$

$\Downarrow$ `LF(T)`

$\texttt{leaf\_repr(T\#)} = \langle\langle 14, 15\rangle, \mathbf{1}\rangle$

Figure 2: Running example for Algorithm 2, finding leaf-type LCP values. **Left**. The BWT matrix and LCP array of the string collection $\{AAGCT\#, CTATA\#, GATAT\#\}$. **Top right**. Suppose we pop from the stack the representation of the leaf #, of string depth 0 (we do not count terminators in the string depth). The range of the leaf spans rows 1-3 (in orange). Then, by Algorithm 2 the LCP value of the positions inside the leaf's range (except the first) is $0 = |\#| - 1$ (in blue). We therefore output the LCP pairs $\langle 2, 0\rangle$ and $\langle 3, 0\rangle$. **Bottom right**. After left-extending # with T by applying the LF mapping to the Ts in the BWT range 1-3 (in red), we obtain the range 14-15 of the leaf $T\#$. By Algorithm 2, the LCP value of position 15 is therefore $1 = |T\#| - 1$ (in green) and we output the LCP pair $\langle 15, 1\rangle$.

It is easy to see that after this step all and only the node-type LCP values equal to 0 have been correctly computed. Assume, by induction, that all node-type LCP values less than or equal to ℓ have been correctly output, and that we are about to extract from the queue the first triple $\langle L, R, \ell+1\rangle$ having length $\ell+1$. For each extracted triple with length $\ell+1$ associated to a string W, consider the triple $\langle L', R', \ell+2\rangle$ associated to one of its left-extensions cW. If $LCP[R'+1]$ has been computed, i.e. if $U[R'+1] = 1$, then we have nothing to do. However, if $U[R'+1] = 0$, then it must be the case that (i) the corresponding LCP value satisfies $LCP[R'+1] \geq \ell+1$, since by induction we have already computed all node-type LCP values smaller than or equal to ℓ, and (ii) $LCP[R'+1]$ is of node-type, since otherwise the BWT interval of cW would also include position $R'+1$. On the other hand, it cannot be the case that $LCP[R'+1] > \ell+1$ since otherwise the cW-interval would include position $R'+1$. We therefore conclude that $LCP[R'+1] = \ell+1$ must hold.

Completeness - Algorithm 1. The above argument settles correctness. To prove completeness, assume that, at some point, $U[i] = 0$ and the value of $LCP[i]$ to be computed and output is $\ell+1$. We want to show that we will pull a triple $\langle L, R, \ell+1\rangle$ from the queue corresponding to a string W (note that $\ell+1 = |W|$ and, moreover, W could end with #) such that one of the left-extensions aW of W satisfies $\texttt{range}(\texttt{aW}) = \langle L', i-1\rangle$, for some L'. This will show that, at some point, we will output the LCP pair $(i, \ell+1)$. We proceed by induction on $|W|$. Note that we separately output all LCP values equal to 0. The base case $|W| = 1$ is easy: by the way we initialized the queue, $\langle \texttt{range}(\texttt{c}), 1\rangle$, for all $c \in \Sigma$, are the first triples we pop. Since we left-extend these ranges with all alphabet's characters except #, it is easy to see that all LCP values equal to 1 have been output. From now on we can therefore assume that we are working on LCP values equal to $\ell+1 > 1$, i.e. $W = b \cdot V$, for $b \in \Sigma - \{\#\}$ and $V \in \Sigma^+$. Let abV be the length-$(\ell+2)$ left-extension of $W = bV$ such that $\texttt{right}(\texttt{abV}) + \texttt{1} = i$. Since, by our initial hypothesis, $\texttt{LCP}[\texttt{i}] = \ell+1$, the collection contains also a suffix aU lexicographically larger than abV and such that $\texttt{LCP}(\texttt{aU}, \texttt{abV}) = \ell+1$. But then, it must be the case that $\texttt{LCP}(\texttt{right}(\texttt{bV}) + \texttt{1}) = \ell$ (it cannot be smaller by the existence of U and it cannot be larger since $|bV| = \ell+1$). By inductive hypothesis, this value was set after popping a triple $\langle L'', R'', \ell\rangle$ corresponding to string V, left-extending V with b, and pushing $\langle \texttt{range}(\texttt{bV}), \ell+1\rangle$ in the queue. This ends the completeness proof since we showed that $\langle \texttt{range}(\texttt{bV}), \ell+1\rangle$ is in the queue, so at some point we will pop it, extend it with a, and output $(right(abV)+1, \ell+1) = (i, \ell+1)$. If the queue uses too much space, then Algorithm 1 switches to a stack and Lines 4-15 are executed instead of Line 2. Note that this pseudocode fragment corresponds to Belazzougui's enumeration algorithm, except that now we also set LCP values in Line 10. By the enumeration procedure's correctness, we have that, in Line 10, $\langle \texttt{first}_\texttt{W}[\texttt{1}], \texttt{first}_\texttt{W}[\texttt{t}+\texttt{1}]\rangle$ is the SA-range of a right-maximal string W with $\ell = |W|$, and $\texttt{first}_\texttt{W}[\texttt{i}]$ is the first position of the SA-range of Wc_i, with $i = 1, \ldots, t$, where $c_1, \ldots, c_2$ are all the (sorted) right-extensions of W. Then, clearly each LCP value in Line 10 is of node-type and has value ℓ, since it is the LCP between two strings prefixed by $W \cdot \texttt{chars}_\texttt{W}[\texttt{i} - \texttt{1}]$ and $W \cdot \texttt{chars}_\texttt{W}[\texttt{i}]$.

Similarly, completeness of the procedure follows from the completeness of the enumeration algorithm. Let $LCP[i]$ be of node-type. Consider the prefix Wb of length $LCP[i]+1$ of the i-th suffix in the lexicographic ordering of all strings' suffixes. Since $LCP[i] = |W|$, the $(i-1)$-th suffix is of the form Wa, with $b \neq a$, and W is right-maximal. But then, at some point our enumeration algorithm will visit the representation of W, with $|W| = \ell$. Since i is the first position of the range of Wb, we have that $i = \texttt{first}_\texttt{W}[\texttt{j}]$ for some $j \geq 2$, and Line 10 correctly outputs the LCP pair $(first_W[j], |W|) = (i, |W|)$.

Correctness and completeness - Algorithm 2. To prove the correctness and completeness of this procedure, it is sufficient to note that the `while` loop iterates over all ranges $\langle L, R\rangle$ of strings ending with # and not containing # anywhere else (note that we start from the range of # and we proceed by recursively left-extending this range with symbols different than #). Then, for each such range we conclude that $LCP[L+1, R]$ is equal to ℓ, i.e. the string depth of the corresponding string (excluding the final character #). By their definition, all leaf-type LCP values are correctly computed in this way.

Complexity - Algorithm 1. If $\sigma > \sqrt{n}/\log^2 n$, then we run Beller et al's algorithm, which terminates in $O(n \log \sigma)$ time and uses $O(n) = o(n \log \sigma)$ bits of additional working space. Otherwise, we perform a linear number of operations on the stack since, as observed in Section 3, the number of Weiner links is linear. By the same analysis of Section 3, the operation in Line 12 takes $O(k \log \sigma)$ amortized time on wavelet trees, and sorting in Line 13 (using any comparison-sorting algorithm sorting m integers in $O(m \log m)$ time) takes $O(k \log \sigma)$ time. Note that in this sorting step we can afford storing in temporary space nodes $x_1, \ldots, x_k$ since this takes additional space $O(k\sigma \log n) = O(\sigma^2 \log n) = O(n/\log^3 n) = o(n)$ bits. All these operations sum up to $O(n \log \sigma)$ time. Since the stack always takes at most $O(\sigma^2 \log^2 n)$ bits and $\sigma \leq \sqrt{n}/\log^2 n$, the stack's size never exceeds $O(n/\log^2 n) = o(n)$ bits.

Complexity - Algorithm 2. Note that, in the `while` loop, we start from the interval of # and recursively left-extend with characters different than # until this is possible. It follows that we visit the intervals of all strings of the form $W\#$ such that # does not appear inside W. Since these intervals form a cover of $[1, n]$, their number (and therefore the number of iterations in the `while` loop) is also bounded by n. This is also the maximum number of operations performed on the queue/stack. Using Beller et al.'s implementation for the queue and a simple vector for the stack, each operation takes constant amortized time. Operating on the stack/queue takes therefore overall $O(n)$ time. For each interval $\langle L, R\rangle$ popped from the queue/stack, in Line 13 we output $R - L - 2$ LCP values. As observed above, these intervals form a cover of $[1, n]$ and therefore Line 13 is executed no more than n times. Line 18 takes time $O(k \log \sigma)$. Finally, in Line 19 we sort at most σ intervals. Using any fast comparison-based sorting algorithm, this costs overall at most $O(n \log \sigma)$ time.

As far as the space usage of Algorithm 2 is concerned, note that we always push just pairs interval/length ($O(\log n)$ bits) in the queue/stack. If $\sigma > n/\log^3 n$, we use Beller et al.'s queue, taking at most $O(n) = o(n \log \sigma)$ bits of space. Otherwise, the stack's size never exceeds $O(\sigma \cdot \log n)$ elements, with

each element taking $O(\log n)$ bits. This amounts to $O(\sigma \cdot \log^2 n) = O(n/\log n) = o(n)$ bits of space usage. Moreover, in Lines 18-19 it holds $\sigma \leq n/\log^3 n$ so we can afford storing temporarily all intervals returned by `getIntervals` in $O(k \log n) = O(\sigma \log n) = O(n/\log^2 n) = o(n)$ bits. □

Combining Lemma 2 and Lemma 1, we obtain:

Theorem 3. *Given the word-packed Burrows-Wheeler Transform of a collection $\mathcal{S} = \{T_1, \ldots, T_m\}$ of total length n on alphabet $[1, \sigma]$, we can build the LCP array of the collection in $O(n \log \sigma)$ time using $o(n \log \sigma)$ bits of working space on top of the BWT.*

6. Enumerating Suffix Tree Intervals

In this section we show that the procedures described in Section 5 can be used to enumerate all suffix tree intervals—that is, the suffix array intervals of all right-maximal text substrings—taking as input the BWT of a text. Note that in this section we consider just simple texts rather than string collections as later we will use this procedure to build the compressed suffix tree of a text.

When $\sigma \leq \sqrt{n}/\log^2 n$, we can directly use Belazzougui's procedure (Theorem 1), which already solves the problem. When $\sigma > \sqrt{n}/\log^2 n$, we modify Beller et al.'s procedure (Theorem 2) to enumerate suffix tree intervals using $O(n) = o(n \log \sigma)$ bits of working space, as follows.

We recall that (see Section 4), Beller et al's procedure can be conveniently described using two separate queues: Q_ℓ and $Q_{\ell+1}$. At each step, we pop from Q_ℓ an element $\langle \langle L, R \rangle, |W| \rangle$ with $\langle L, R \rangle = \texttt{range}(\texttt{W})$ and $|W| = \ell$ for some string W, left-extend the range with all $a \in \texttt{BWT.rangeDistinct}(\texttt{L}, \texttt{R})$, obtaining the ranges $\texttt{range}(\texttt{aW}) = \langle L_a, R_a \rangle$ and, only if $U[R_a + 1] = 0$, set $U[R_a + 1] \leftarrow 1$, output the LCP pair $(R_a + 1, |W|)$, and push $\langle \langle L_a, R_a \rangle, |W| + 1 \rangle$ into $Q_{\ell+1}$. Note that, since $LCP[R_a + 1] = |W|$ we have that the R_a-th and $(R_a + 1)$-th smallest suffixes start, respectively, with aXc and aXd for some $c < d \in \Sigma$, where $W = Xc$. This implies that aX is right-maximal. It is also clear that, from the completeness of Beller et al.'s procedure, all right-maximal text substrings are visited by the procedure, since otherwise the LCP values equal to $\ell = |aX|$ inside $\texttt{range}(\texttt{aX})$ would not be generated. Note that the procedure generates only intervals of right-maximal substrings, thus intervals corresponding to suffix tree leaves are not generated. However, these intervals are $\langle \texttt{i}, \texttt{i} \rangle$ for all $1 \leq i \leq n$, therefore they can be generated before starting the procedure. It follows that, in order to generate the suffix tree intervals of all suffix tree nodes (leaves included) *once*, we need two extra ingredients: (i) whenever we pop from Q_ℓ an element $\langle \langle L, R \rangle, |W| \rangle$ corresponding to a string $W = Xc$, we also need the range of X, and (ii) we need to quickly check if a given range $\texttt{range}(\texttt{aX})$ of a right-maximal substring aX has already been output. Point (ii) is necessary since, using only the above procedure (augmented with point (i)), $\texttt{range}(\texttt{aX})$ will be output for

each of its right-extensions (except the lexicographically largest, which does not cause the generation of an LCP pair).

Remember that, in order to keep space usage under control (i.e. $O(n)$ bits), we represent Q_ℓ as a standard queue of pairs $\langle \texttt{range(W)}, |W| \rangle$ if and only if $|Q_\ell| < n/\log n$. For now, let us assume that the queue size does not exceed this quantity (the other case will be considered later). In this case, to implement point (i) we simply augment queue pairs as $\langle \texttt{range(W)}, \texttt{range(X)}, |W| \rangle$, where $W = Xc$ for some $c \in \Sigma$. When left-extending W with a character a, we also left-extend X with a, obtaining $\texttt{range(aX)}$. Let $\texttt{range(aW)} = \langle L_a, R_a \rangle$. At this point, if $R_a < n$ and $U[R_a + 1] = 0$ we do the following:

1. we set $U[R_a + 1] \leftarrow 1$,
2. we push $\langle \texttt{range(aW)}, \texttt{range(aX)}, |W| + 1 \rangle$ in $Q_{\ell+1}$, and
3. if $\texttt{range(aX)}$ has not already been generated, we output the suffix tree range $\texttt{range(aX)}$.

Note that steps (1) and (2) correspond to Beller et al.'s procedure. By the way we defined our procedure, we add an additional small difference with their algorithm: instead of initializing the queue with the range of $W = \epsilon$ (for which X would not be defined), we start by outputting the suffix tree range $\langle 1, n \rangle$ (that is, the range of the root) and by initializing the queue with the elements $\langle \texttt{range(c)}, \texttt{range}(\epsilon), \texttt{1} \rangle$ for each $c \in \Sigma$. Since this corresponds (in Beller et al.'s algorithm) to having generated all LCP values equal to 0, we also need to set $U[L_c] = 1$ for all the ranges $\langle L_c, R_c \rangle = \texttt{range(c)}$, $c \in \Sigma$. Figure 3 shows an example of our full procedure.

The test in step (3) can be implemented as follows. Note that a suffix array range $\texttt{range(aX)} = \langle L, R \rangle$ can be identified unambiguously by the two integers L and $|aX| = \ell$. Note also that we generate suffix tree intervals in increasing order of string depth (i.e. when popping elements from Q_ℓ, we output suffix array intervals of string depth ℓ). It follows that we can keep a bitvector GEN_ℓ of length n recording in $GEN_\ell[i]$ whether or not the suffix array interval of the string of length ℓ whose first coordinate is i has already been output. Each time we change the value of a bit $GEN_\ell[i]$ from 0 to 1, we also push i into a stack SET_ℓ. Let us assume for now that also SET_ℓ's size does not exceed $n/\log n$ (later we will consider a different representation for the other case). Then, also the bit-size of SET_ℓ will never exceed $O(n)$ bits. After Q_ℓ has been emptied, for each $i \in SET_\ell$ we set $GEN_\ell[i] \leftarrow 0$. This makes all GEN_ℓ's entries equal to 0, and we can thus re-use its space for $GEN_{\ell+1}$ at the next stage (i.e. when popping elements from $Q_{\ell+1}$).

Now, let us consider the case $|Q_\ell| \geq n/\log n$. The key observation is that Q_ℓ exceeds this value for at most $O(\log n)$ values of ℓ, therefore we can afford spending extra $O(n/\log n)$ time to process each of these queues. As seen in Section 4 (paragraph "Queue implementation"), whenever Q_ℓ's size exceeds $n/\log n$ (while pushing elements in it) we switch to a different queue representation using packed bitvectors. Point (i) can be solved by storing two additional bitvectors as follows. Suppose we are about to push the triple $\langle \texttt{range(W)}, \texttt{range(X)}, |W| \rangle$ in Q_ℓ, where $W = Xc$ for some $c \in \Sigma$. The solution seen in Section 4 consisted in

index	BWT	Suffixes
1	A	#
2	T	A #
3	T	A T A #
4	#	C T A T A #
5	A	T A #
6	C	T A T A #

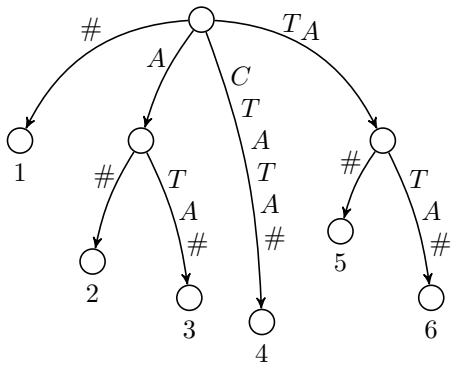

Figure 3: Running example of our extension of Beller et al.'s algorithm for generating the suffix tree nodes' intervals (leaves included). In the example, the string is $CTATA\#$. **Left**. BWT and sorted suffixes of the string. **Right.** Suffix tree of the string. First, we output the leaves' intervals, $\langle \mathbf{i}, \mathbf{i} \rangle$ for $i = 1, \ldots, 6$, and the root's interval, $\langle \mathbf{1}, \mathbf{6} \rangle$. We initialize the queue with triples $\langle \texttt{range(c)}, \texttt{range}(\epsilon), 1 \rangle$ for all $c \in \Sigma$: $\mathcal{Q} = \{\langle\langle 1,1\rangle, \langle 1,6\rangle, 1\rangle, \langle\langle 2,3\rangle, \langle 1,6\rangle, 1\rangle, \langle\langle 4,4\rangle, \langle 1,6\rangle, 1\rangle, \langle\langle 5,6\rangle, \langle 1,6\rangle, 1\rangle\}$, and we create the bitvector $U[1,6] = 1,1,0,1,1,0$ (remember that we set all bits corresponding to the beginning of the range of a single character). The first element we pop from the queue is $\langle\langle 1,1\rangle, \langle 1,6\rangle, 1\rangle$, corresponding to the strings $W = \#$ and $X = \epsilon$. There is only one way to left-extend W, namely, with letter A. The range of $A\#$ is on rows $\langle L, R\rangle = \langle 2,2\rangle$; since $U[R+1] = U[3] = 0$, we (1) set $U[3] \leftarrow 1$, (2) append the left-extended intervals $\langle\langle 2,2\rangle, \langle 2,3\rangle, 2\rangle$ (of strings $A\#$ and A, respectively) at the end of the queue, and, since $\texttt{range(X)} = \texttt{range(A)} = \langle \mathbf{2}, \mathbf{3} \rangle$ has not been output yet, we (3) output the suffix tree range $\langle \mathbf{2}, \mathbf{3} \rangle$ of the right-maximal string A. The queue is now $\mathcal{Q} = \{\langle\langle 2,3\rangle, \langle 1,6\rangle, 1\rangle, \langle\langle 4,4\rangle, \langle 1,6\rangle, 1\rangle, \langle\langle 5,6\rangle, \langle 1,6\rangle, 1\rangle, \langle\langle 2,2\rangle, \langle 2,3\rangle, 2\rangle\}$. We extract the first element, corresponding to $W = A$ and $X = \epsilon$. W can only be left-extended with T. The resulting range of TA is $\langle 5,6\rangle$; since $6 = n$ however, we do not proceed further. Also the next two elements, corresponding to $W = C$ and $W = T$, are discarded since all their extensions cW, with $\texttt{range(cW)} = \langle L, R\rangle$ are such that $U[R+1] = 1$. The only element left in the queue is $\langle\langle 2,2\rangle, \langle 2,3\rangle, 2\rangle$, corresponding to $W = A\#$ and $X = A$. The string $A\#$ can be extended only by character T. This leads to the range $\texttt{range(TA\#)} = \langle L, R\rangle = \langle 5,5\rangle$, for which we have $U[R+1] = U[6] = 0$. We therefore set $U[6] \leftarrow 1$ and output the suffix tree interval of string $TX = TA$, that is, $\langle \mathbf{5}, \mathbf{6} \rangle$. The algorithm continues by appending to the queue the intervals of TW and TX but will not output any other suffix tree interval since all bits in U are now set.

marking, in two packed bitvectors $\texttt{open}[1,\texttt{n}]$ and $\texttt{close}[1,\texttt{n}]$, the start and end points of $\texttt{range}(\texttt{W})$. Now, we just use two additional packed bitvectors $\overline{\texttt{open}}[1,\texttt{n}]$ and $\overline{\texttt{close}}[1,\texttt{n}]$ to also mark the start and end points of $\texttt{range}(\texttt{X})$. As seen in Section 4 (paragraph "Queue implementation"), intervals are extracted from Q_ℓ by scanning $\texttt{open}[1,\texttt{n}]$ and $\texttt{close}[1,\texttt{n}]$ in $O(n/\log n+|Q_\ell|)$ time. Note that W is a right-extension of X, therefore $\texttt{range}(\texttt{W})$ is contained in $\texttt{range}(\texttt{X})$. It follows that we can scan in parallel the bitvectors $\texttt{open}[1,\texttt{n}]$, $\texttt{close}[1,\texttt{n}]$, $\overline{\texttt{open}}[1,\texttt{n}]$, and $\overline{\texttt{close}}[1,\texttt{n}]$ and retrieve, for each $\texttt{range}(\texttt{W})$ extracted from the former two bitvectors, the (unique in the queue) interval $\texttt{range}(\texttt{X})$ enclosing $\texttt{range}(\texttt{W})$ (using the latter two bitvectors). More formally, whenever finding a bit set at $\overline{\texttt{open}}[\texttt{i}]$, we search $\overline{\texttt{close}}[\texttt{i},\texttt{n}]$ to find the next bit set. Let us call j the position containing such bit set. Then, we similarly scan $\texttt{open}[\texttt{i},\texttt{j}]$ and $\texttt{close}[\texttt{i},\texttt{j}]$ to generate all intervals $\langle l,r\rangle$ enclosed by $\langle i,j\rangle$, and for each of them generate the triple $\langle\langle l,r\rangle,\langle i,j\rangle,\ell\rangle$. Again, exploiting word-parallelism the process takes $O(n/\log n+|Q_\ell|)$ time to extract all triples $\langle\texttt{range}(\texttt{W}),\texttt{range}(\texttt{X}),|W|\rangle$ from Q_ℓ.

A similar solution can be used to solve point (ii) for large SET_ℓ. Whenever SET_ℓ exceeds size $n/\log n$, we simply empty it and just use bitvector GEN_ℓ. This time, however, this bitvector is packed in $O(n/\log n)$ words. It can therefore be erased (i.e. setting all its entries to 0) in $O(n/\log n)$ time, and we do not need to use the stack SET_ℓ at all. Since (a) we insert an element in some SET_ℓ only when outputting a suffix tree range and (b) in total we output $O(n)$ such ranges, SET_ℓ can exceed size $n/\log n$ for at most $O(\log n)$ values of ℓ. We conclude that also the cost of creating and processing all GEN_ℓ and SET_ℓ amortizes to $O(n)$.

To sum up, the overall procedure runs in $O(n\log\sigma)$ time and uses $O(n)$ bits of space. By combining it with Belazzougui's procedure as seen above (i.e. choosing the right procedure according to the alphabet's size), we obtain:

Lemma 4. *Given a wavelet tree representing the Burrows-Wheeler transform of a text T of length n on alphabet $[1,\sigma]$, in $O(n\log\sigma)$ time and $o(n\log\sigma)$ bits of working space we can enumerate the suffix array intervals corresponding to all right maximal text's substrings.*

7. Building the PLCP Bitvector

The PLCP array is defined as $PLCP[i]=LCP[ISA[i]]$, and can thus be used to retrieve LCP values as $LCP[i]=PLCP[SA[i]]$ (note that this requires accessing the suffix array). Kasai et al. showed in [16] that PLCP is almost increasing: $PLCP[i+1]\geq PLCP[i]-1$. This allows representing it in small space as follows. Let $\texttt{plcp}[1,\texttt{2n}]$ denote the bitvector having a bit set at each position $PLCP[i]+2i$, for $i=1,\ldots,n$ (and 0 in all other positions). Since $PLCP[i+1]\geq PLCP[i]-1$, the quantity $PLCP[i]+2i$ is different for each i. By definition, $PLCP[i]$ can be written as $j-2i$, where j is the position of the i-th bit set in $\texttt{plcp}$; this shows that each PLCP entry can be retrieved in

constant time using the bitvector `plcp`, augmented to support constant-time *select* queries.

We now show how to build the `plcp` bitvector in small space using the LCP enumeration procedure of Section 5. Our procedure relies on the concept of *irreducible LCP values*:

Definition 1. $LCP[i]$ is said to be *irreducible* if and only if either $i = 0$ or $BWT[i] \neq BWT[i-1]$ hold.

We call *reducible* a non-irreducible LCP value. We extend the above definition to PLCP values, saying that $PLCP[i]$ is irreducible if and only if $LCP[ISA[i]]$ is irreducible. The following Lemma, shown in [42], is easy to prove (see also [25, Lem. 4]):

Lemma 5 ([42], Lem. 1)**.** *If $PLCP[i]$ is reducible, then $PLCP[i] = PLCP[i-1] - 1$.*

We also make use of the following Theorem from Kärkkäinen et al. [25]:

Theorem 4 ([25], Thm. 1)**.** *The sum of all irreducible lcp values is at most $2n \log n$.*

Our strategy is as follows. We divide $BWT[1, n]$ in $\lceil n/B \rceil$ blocks $BWT[(i-1) \cdot B + 1, i \cdot B]$, $i = 1, \ldots, \lceil n/B \rceil$ of size B (assume for simplicity that B divides n). For each block $i = 1, \ldots, \lceil n/B \rceil$, we use Lemma 2 to enumerate all pairs $(j, LCP[j])$. Whenever we generate a pair $(j, LCP[j])$ such that (i) j falls in the current block's range $[(i-1) \cdot B + 1, i \cdot B]$, (ii) $LCP[j] > \log^3 n$, and (iii) $LCP[j]$ is irreducible (this can be checked easily using Definition 1), we store $(j, LCP[j])$ in a temporary array `LARGE_LCP` (note: each such pair requires $O(\log n)$ bits to be stored). By Theorem 4, there cannot be more than $2n/\log^2 n$ irreducible LCP values being larger than $\log^3 n$, that is, `LARGE_LCP` will never contain more than $2n/\log^2 n$ values and its bit-size will never exceed $O(n/\log n) = o(n)$ bits. We also mark all such relative positions $j - (i-1) \cdot B$ in a bitvector of length B with rank support and radix-sort `LARGE_LCP` in $O(B)$ time to guarantee constant-time access to $LCP[j]$ whenever conditions (i-iii) hold true for index j. On the other hand, if (i) j falls in the current block's range $[(i-1) \cdot B + 1, i \cdot B]$, (ii) $LCP[j] \leq \log^3 n$, and (iii) $LCP[j]$ is irreducible then we can store $LCP[j]$ in another temporary vector `SMALL_LCP[1, B]` as follows: `SMALL_LCP`$[j-(i-1) \cdot B] \leftarrow LCP[j]$ (at the beginning, the vector is initialized with undefined values). By condition (ii), `SMALL_LCP` can be stored in $O(B \log\log n)$ bits. Using `LARGE_LCP` and `SMALL_LCP`, we can access in constant time all irreducible values $LCP[j]$ whenever j falls in the current block $[(i-1) \cdot B + 1, i \cdot B]$. At this point, we enumerate all pairs $(i, ISA[i])$ in text order (i.e. for $i = 1, \ldots, n$) using the FL function on the BWT. Whenever one of those pairs $(i, ISA[i]) = (i, j)$ is such that (i) j falls in the current block's range $[(i-1) \cdot B + 1, i \cdot B]$ and

(ii) $LCP[j]$ is irreducible, we retrieve $LCP[j]$ in constant time as seen above and we set $\texttt{plcp}[\texttt{2i} + \texttt{LCP[j]}] \leftarrow 1$; the correctness of this assignment follows from the fact that $j = ISA[i]$, thus $LCP[j] = PLCP[i]$. Using Lemma 5, we can moreover compute the reducible PLCP values that follow $PLCP[i]$ in text order (up to the next irreducible value), and set the corresponding bits in `plcp`. After repeating the above procedure for all blocks $BWT[(i-1) \cdot B + 1, i \cdot B]$, $i = 1, \ldots, \lceil n/B \rceil$, we terminate the computation of bitvector `plcp`. For each block, we spend $O(n \log \sigma)$ time (one application of Lemma 2 and one BWT navigation to generate all pairs $(i, ISA[i])$). We also spend $O(n/\log^2 n)$ time to allocate the instances of `LARGE_LCP` across all blocks. Overall, we spend $O((n^2/B) \log \sigma + n \log \sigma)$ time across all blocks. The space used is $o(n) + O(B \cdot \log \log n)$ bits on top of the BWT. By setting $B = (\epsilon \cdot n \log \sigma)/\log \log n$ we obtain our result:

Lemma 6. *Given a wavelet tree for the Burrows-Wheeler transform of a text T of length n on alphabet $[1, \sigma]$, for any parameter $0 < \epsilon \leq 1$ we can build the PLCP bitvector in $O(n(\log \sigma + \epsilon^{-1} \log \log n))$ time and $\epsilon \cdot n \log \sigma + o(n)$ bits of working space on top of the input BWT and the optput.*

8. Building the Suffix Tree Topology

In order to build the suffix tree topology we use a strategy analogous to the one proposed by Belazzougui [24]. The main observation is that, given a procedure that enumerates suffix tree intervals, for each interval $[l, r]$ we can increment a counter $\texttt{Open[l]}$ and a counter $\texttt{Close[r]}$, where `Open` and `Close` are integer vectors of length n. Then, the BPS representation of the suffix tree topology can be built by scanning left-to right the two arrays and, for each $i = 1, \ldots, n$, append $\texttt{Open[i]}$ open parentheses followed by $\texttt{Close[i]}$ close parentheses to the BPS representation. The main drawback of this solution is that it takes too much space: $2n \log n$ bits to store the two arrays. Belazzougui solves this problem by noticing that the sum of all the values in the two arrays is the length of the final BPS representation, that is, at most $4n$. This makes it possible to represent the arrays in just $O(n)$ bits of space by representing (the few) large counters in plain form and (the many) small counters using delta encoding (while still supporting updates in constant time).

Our goal in this section is to reduce the working space from $O(n)$ to a (small) fraction of $n \log \sigma$. A first idea could be to iterate Belazzougui's strategy on chunks of the interval $[1, n]$. Unfortunately, this does not immediately give the correct solution as a chunk could still account for up to $\Theta(n)$ parentheses, no matter what the length of the chunk is; as a result, Belazzougui's representation could still take $O(n)$ bits of space for a chunk (when using large enough chunks to keep the running time under control as seen in the previous section). We use a solution analogous to the one discussed in the previous section. This solution corresponds to the first part of Belazzougui's strategy (in particular, we will store small counters in plain form instead of using delta

encoding). We divide $BWT[1,n]$ in $\lceil n/B \rceil$ blocks $BWT[(i-1)\cdot B+1, i\cdot B]$, $i = 1,\ldots,\lceil n/B \rceil$ of size B (assume for simplicity that B divides n). For each block $i = 1,\ldots,\lceil n/B \rceil$, we use Lemma 4 to enumerate all suffix tree intervals $[l,r]$. We keep two arrays $\texttt{Open}[1,\texttt{B}]$ and $\texttt{Close}[1,\texttt{B}]$ storing integers of $2\log\log n$ bits each. Whenever the beginning l of a suffix tree interval $[l,r]$ falls inside the current block $[(i-1)\cdot B+1, i\cdot B]$, we increment $\texttt{Open}[\texttt{l} - (\texttt{i} - 1)\cdot \texttt{B}]$ (the description is analogous for index r and array `Close`). If $\texttt{Open}[\texttt{l} - (\texttt{i} - 1)\cdot \texttt{B}]$ reaches the maximum value $2^{2\log\log n - 1}$, we no longer increment it. Adopting Belazzougui's terminology, we call such a bucket "saturated". After having generated all suffix tree intervals, let k be the number of saturated counters. We allocate a vector `LARGE_COUNTERS` storing k integers of $\log n + 2$ bits each (enough to store the value $4n$, i.e. an upper-bound to the value that a counter can reach). We also allocate a bitvector of length B marking saturated counters, and process it to support constant-time rank queries. This allows us to obtain in constant time the location in `LARGE_COUNTERS` corresponding to any saturated counter in the block. We generate all suffix tree intervals for a second time using again Lemma 4, this time incrementing (in `LARGE_COUNTERS`) only locations corresponding to saturated counters. Since the BPS sequence has length at most $4n$ and a counter saturates when it reaches value $\Theta(\log^2 n)$, we have that $k = O(n/\log^2 n)$ and thus `LARGE_COUNTERS` takes at most $O(n/\log n) = o(n)$ bits to be stored. The rest of the analysis is identical to the algorithm described in the previous section. For each block, we spend $O(n\log\sigma)$ time (two applications of Lemma 4). We also spend $O(n/\log^2 n)$ time to allocate the instances of `LARGE_COUNTERS` across all blocks. Overall, we spend $O((n^2/B)\log\sigma + n\log\sigma)$ time across all blocks. The space used is $o(n) + O(B\cdot\log\log n)$ bits on top of the BWT. By setting $B = (\epsilon\cdot n\log\sigma)/\log\log n$ we obtain:

Lemma 7. *Given a wavelet tree for the Burrows-Wheeler transform of a text T of length n on alphabet $[1,\sigma]$, for any parameter $0 < \epsilon \leq 1$ we can build the BPS representation of the suffix tree topology in $O(n(\log\sigma + \epsilon^{-1}\log\log n))$ time and $\epsilon\cdot n\log\sigma + o(n)$ bits of working space on top of the input BWT and the optput.*

To conclude, we note that our procedures can be immediately used to build space-efficiently the compressed suffix tree described by Sadakane [18] starting from the BWT. The only missing ingredients are (i) to augment the BWT with a suffix array sample in order to turn it into a CSA, and (ii) to preprocess the PLCP and BPS sequences to support fast queries (*select* on the PLCP and navigational queries on the BPS). Step (i) can be easily performed in $O(n\log\sigma)$ time and $n + o(n)$ bits of working space with a folklore solution that iteratively applies function LF to navigate all BWT's positions and collect one suffix array sample every $O(\log^{1+\delta} n/\log\sigma)$ text positions, for any fixed $\delta > 0$ (using a succinct bitvector to mark sampled positions). The resulting CSA takes $n\log\sigma + o(n\log\sigma)$ bits of space and allows computing any $SA[i]$ in $O(\log^{1+\delta} n)$ time. Step (ii) can be performed in $O(n)$ time and $o(n)$ bits of working space using textbook solutions (see [43]). Combining this with Lemmas 1, 6, and 7, we obtain:

Theorem 5. *Given the re-writable word-packed BWT of a text T of length n on alphabet $[1, \sigma]$, for any parameter $0 < \epsilon \leq 1$ we can replace it in $O(n(\log \sigma + \epsilon^{-1} \log\log n))$ time and $(\epsilon + o(1)) \cdot n \log \sigma$ bits of working space with a compressed suffix tree taking $n \log \sigma + 6n + o(n \log \sigma)$ bits of space and supporting all operations in $O(\texttt{polylog n})$ time.*

9. Merging BWTs in Small Space

In this section we use our space-efficient BWT-navigation strategies to tackle an additional problem: to merge the BWTs of two string collections. In [24, 32], Belazzougui et al. show that Theorem 1 can be adapted to merge the BWTs of two texts T_1, T_2 and obtain the BWT of the collection $\{T_1, T_2\}$ in $O(nk)$ time and $n \log \sigma(1 + 1/k) + 11n + o(n)$ bits of working space for any $k \geq 1$ [32, Thm. 7]. We show that our strategy enables a more space-efficient algorithm for the task of merging BWTs of collections. The following theorem, whose proof is reported later in this section, merges two BWTs by computing the binary DA of their union. After that, the merged BWT can be streamed to external memory (the DA tells how to interleave characters from the input BWTs) and does not take additional space in internal memory. Similarly to what we did in the proof of Theorem 3, this time we re-use the space of the Document Array to accommodate the extra n bits needed to replace the BWTs of the two collections with their wavelet matrices. This is the main result of this section:

Theorem 6. *Given the Burrows-Wheeler Transforms of two collections $\mathcal{S}_1$ and $\mathcal{S}_2$ of total length n on alphabet $[1, \sigma]$, we can compute the Document Array of $\mathcal{S}_1 \cup \mathcal{S}_2$ in $O(n \log \sigma)$ time using $o(n \log \sigma)$ bits of working space on top of the input BWTs and the output DA.*

We also briefly discuss how to extend Theorem 6 to build the LCP array of the merged collection. In Section 10 we present an implementation of our algorithms and an experimental comparison with `eGap` [44], the state-of-the-art tool designed for the same task of merging BWTs while inducing the LCP of their union.

The procedure of Algorithm 2 can be extended to merge BWTs of two collections $\mathcal{S}_1$, $\mathcal{S}_2$ using $o(n \log \sigma)$ bits of working space on top of the input BWTs and output Document Array (here, n is the cumulative length of the two BWTs). The idea is to simulate a navigation of the *leaves* of the generalized suffix tree of $\mathcal{S}_1 \cup \mathcal{S}_2$ (note: for us, a collection is an ordered multi-set of strings). Our procedure differs from that described in [32, Thm. 7] in two ways. First, they navigate a subset of the suffix tree *nodes* (so-called *impure* nodes, i.e. the roots of subtrees containing suffixes from distinct strings), whereas we navigate leaves. Second, their visit is implemented by following Weiner links. This forces them to represent the nodes with the "heavy" representation `repr` of Section 3, which is not efficient on large alphabets. On the contrary, leaves can be represented simply as ranges and allow for a more space-efficient queue/stack representation.

We represent each leaf by a pair of intervals, respectively on $BWT(\mathcal{S}_1)$ and $BWT(\mathcal{S}_2)$, of strings of the form $W\#$. Note that: (i) the suffix array of $\mathcal{S}_1 \cup \mathcal{S}_2$ is covered by the non-overlapping intervals of strings of the form $W\#$, and (ii) for each such string $W\#$, the interval $\texttt{range}(\texttt{W}\#) = \langle L, R\rangle$ in $GSA(\mathcal{S}_1 \cup \mathcal{S}_2)$ can be partitioned as $\langle L, M\rangle \cdot \langle M+1, R\rangle$, where $\langle L, M\rangle$ contains only suffixes from $\mathcal{S}_1$ and $\langle M+1, R\rangle$ contains only suffixes from $\mathcal{S}_2$ (one of these two intervals could be empty). It follows that we can navigate in parallel the leaves of the suffix trees of $\mathcal{S}_1$ and $\mathcal{S}_2$ (using again a stack or a queue containing pairs of intervals on the two BWTs), and fill the Document Array $DA[1, n]$, an array that will tell us whether the i-th entry of $BWT(\mathcal{S}_1 \cup \mathcal{S}_2)$ comes from $BWT(\mathcal{S}_1)$ ($DA[i] = 0$) or $BWT(\mathcal{S}_2)$ ($DA[i] = 1$). To do this, let $\langle L_1, R_1\rangle$ and $\langle L_2, R_2\rangle$ be the ranges on the suffix arrays of $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively, of a suffix $W\#$ of some string in the collections. Note that one of the two intervals could be empty: $R_j < L_j$. In this case, we still require that $L_j - 1$ is the number of suffixes in $\mathcal{S}_j$ that are smaller than $W\#$. Then, in the collection $\mathcal{S}_1 \cup \mathcal{S}_2$ there are $L_1 + L_2 - 2$ suffixes smaller than $W\#$, and $R_1 + R_2$ suffixes smaller than or equal to $W\#$. It follows that the range of $W\#$ in the suffix array of $\mathcal{S}_1 \cup \mathcal{S}_2$ is $\langle L_1 + L_2 - 1, R_1 + R_2\rangle$, where the first $R_1 - L_1 + 1$ entries correspond to suffixes of strings from $\mathcal{S}_1$. Then, we set $DA[L_1 + L_2 - 1, L_2 + R_1 - 1] \leftarrow 0$ and $DA[L_2 + R_1, R_1 + R_2] \leftarrow 1$. The procedure starts from the pair of intervals corresponding to the ranges of the string "#" in the two BWTs, and proceeds recursively by left-extending the current pair of ranges $\langle L_1, R_1\rangle$, $\langle L_2, R_2\rangle$ with the symbols in $\texttt{BWT}_1.\texttt{rangeDistinct}(\texttt{L}_1, \texttt{R}_1) \cup \texttt{BWT}_2.\texttt{rangeDistinct}(\texttt{L}_2, \texttt{R}_2)$. The detailed procedure is reported in Algorithm 3 and we show an example in Figure 4. The leaf visit is implemented, again, using a stack or a queue; this time however, these containers are filled with pairs of intervals $\langle L_1, R_1\rangle$, $\langle L_2, R_2\rangle$. We implement the stack simply as a vector of quadruples $\langle L_1, R_1, L_2, R_2\rangle$. As far as the queue is concerned, some care needs to be taken when representing the pairs of ranges using bitvectors as seen in Section 4 with Beller et al.'s representation. Recall that, at any time, the queue can be partitioned into two sub-sequences associated with LCP values ℓ and $\ell + 1$ (we pop from the former, and push in the latter). This time, we represent each of these two subsequences as a vector of quadruples (pairs of ranges on the two BWTs) as long as the number of quadruples in the sequence does not exceed $n/\log n$. When there are more quadruples than this threshold, we switch to a bitvector representation defined as follows. Let $|BWT(\mathcal{S}_1)| = n_1$, $|BWT(\mathcal{S}_2)| = n_2$, and $|BWT(\mathcal{S}_1 \cup \mathcal{S}_2)| = n = n_1 + n_2$. We keep two bitvectors $\texttt{Open}[1, \texttt{n}]$ and $\texttt{Close}[1, \texttt{n}]$ storing opening and closing parentheses of intervals in $BWT(\mathcal{S}_1 \cup \mathcal{S}_2)$. We moreover keep two bitvectors $\texttt{NonEmpty}_1[1, \texttt{n}]$ and $\texttt{NonEmpty}_2[1, \texttt{n}]$ keeping track, for each i such that $\texttt{Open}[\texttt{i}] = 1$, of whether the interval starting in $BWT(\mathcal{S}_1 \cup \mathcal{S}_2)[i]$ contains suffixes of reads coming from $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively. Finally, we keep four bitvectors $\texttt{Open}_\texttt{j}[1, \texttt{n}_\texttt{j}]$ and $\texttt{Close}_\texttt{j}[1, \texttt{n}_\texttt{j}]$, for $j = 1, 2$, storing non-empty intervals on $BWT(\mathcal{S}_1)$ and $BWT(\mathcal{S}_2)$, respectively. To insert a pair of intervals $\langle L_1, R_1\rangle$, $\langle L_2, R_2\rangle$ in the queue, let $\langle L, R\rangle = \langle L_1 + L_2 - 1, R_1 + R_2\rangle$. We set $\texttt{Open}[\texttt{L}] \leftarrow 1$ and $\texttt{Close}[\texttt{R}] \leftarrow 1$. Then, for $j = 1, 2$, we set $\texttt{NonEmpty}_\texttt{j}[\texttt{L}] \leftarrow 1$, $\texttt{Open}_\texttt{j}[\texttt{L}_\texttt{j}] \leftarrow 1$ and $\texttt{Close}_\texttt{j}[\texttt{R}_\texttt{j}] \leftarrow 1$ if and only if $R_j \geq L_j$.

This queue representation takes $O(n)$ bits. By construction, for each bit set in `Open` at position i, there is a corresponding bit set in $\texttt{Open}_\texttt{j}$ if and only if $\texttt{NonEmpty}_\texttt{j}[\texttt{i}] = 1$ (moreover, corresponding bits set appear in the same order in `Open` and $\texttt{Open}_\texttt{j}$). It follows that a left-to-right scan of these bitvectors is sufficient to identify corresponding intervals on $BWT(\mathcal{S}_1 \cup \mathcal{S}_2)$, $BWT(\mathcal{S}_1)$, and $BWT(\mathcal{S}_2)$. By packing the bits of the bitvectors in words of $\Theta(\log n)$ bits, the t pairs of intervals contained in the queue can be extracted in $O(t+n/\log n)$ time (as described in [27]) by scanning in parallel the bitvectors forming the queue. Particular care needs to be taken only when we find the beginning of an interval $\texttt{Open}[\texttt{L}] = \texttt{1}$ with $\texttt{NonEmpty}_\texttt{1}[\texttt{L}] = 0$ (the case $\texttt{NonEmpty}_\texttt{2}[\texttt{L}] = 0$ is symmetric). Let L_2 be the beginning of the corresponding non-empty interval on $BWT(\mathcal{S}_2)$. Even though we are not storing L_1 (because we only store nonempty intervals), we can retrieve this value as $L_1 = L - L_2 + 1$. Then, the empty interval on $BWT(\mathcal{S}_1)$ is $\langle L_1, L_1 - 1\rangle$.

The same arguments used in the previous section show that the algorithm runs in $O(n \log \sigma)$ time and uses $o(n \log \sigma)$ bits of space on top of the input BWTs and output Document Array. This proves Theorem 6. To conclude, we note that the algorithm can be easily extended to compute the LCP array of the merged collection while merging the BWTs. This requires adapting Algorithm 1 to work on pairs of suffix tree nodes (as we did in Algorithm 3 with pairs of leaves). Results on an implementation of the extended algorithm are discussed in the next section. From the practical point of view, note that it is more advantageous to induce the LCP of the merged collection while merging the BWTs (rather than first merging and then inducing the LCP using the algorithm of the previous section), since leaf-type LCP values can be induced directly while computing the document array.

Note that Algorithm 3 is similar to Algorithm 2, except that now we manipulate pairs of intervals. In Line 27, we sort quadruples according to the length $R_1^i + R_2^i - (L_1^i + L_2^i) + 2$ of the combined interval on $BWT(\mathcal{S}_1 \cup \mathcal{S}_2)$. Finally, note that Backward search can be performed correctly also when the input interval is empty: $\texttt{BWT}_\texttt{j}\texttt{.bwsearch}(\langle \texttt{L}_\texttt{j}, \texttt{L}_\texttt{j} - \texttt{1}\rangle, \texttt{c})$, where $L_j - 1$ is the number of suffixes in $\mathcal{S}_j$ smaller than some string W, correctly returns the pair $\langle L', R'\rangle$ such that $L' - 1$ is the number of suffixes in $\mathcal{S}_j$ smaller than cW: this is true when implementing backward search with a $rank_c$ operation on position L_j; then, if the original interval is empty we just set $R' = L' - 1$ to keep the invariant that $R' - L' + 1$ is the interval's length.

10. Implementation and Experimental Evaluation

We implemented our LCP construction and BWT merge algorithms on DNA alphabet in `https://github.com/nicolaprezza/bwt2lcp` using the language C++. Due to the small alphabet size, it was actually sufficient to implement our extension of Belazzougui's enumeration algorithm (and not the strategy of Beller et al., which becomes competitive only on large alphabets). The repository features a new packed string on DNA alphabet $\Sigma_{DNA} = \{A, C, G, T, \#\}$ using 4 bits per character and able to compute the quintuple $\langle BWT.rank_c(i)\rangle_{i \in \Sigma_{DNA}}$

Algorithm 3 $\texttt{Merge}(\texttt{BWT}_1, \texttt{BWT}_2, \texttt{DA})$

if $\sigma > n/\log^3 n$ **then**
  $\texttt{P} \leftarrow \texttt{new\_queue}()$ ▷ Initialize new queue of interval pairs
else
  $\texttt{P} \leftarrow \texttt{new\_stack}()$ ▷ Initialize new stack of interval pairs
end if
$\texttt{P.push}(\texttt{BWT}_1\texttt{.range}(\#), \texttt{BWT}_2\texttt{.range}(\#))$ ▷ Push SA-ranges of terminator
while not $\texttt{P.empty}()$ **do**
  $\langle L_1, R_1, L_2, R_2 \rangle \leftarrow \texttt{P.pop}()$ ▷ Pop highest-priority element
  for $i = L_1 + L_2 - 1 \ldots L_2 + R_1 - 1$ **do**
    $\texttt{DA}[i] \leftarrow 0$ ▷ Suffixes from $\mathcal{S}_1$
  end for
  for $i = L_2 + R_1 \ldots R_1 + R_2$ **do**
    $\texttt{DA}[i] \leftarrow 1$ ▷ Suffixes from $\mathcal{S}_2$
  end for
  if $\sigma > n/\log^3 n$ **then**
    $\texttt{P.push}(\texttt{getIntervals}(\texttt{L}_1, \texttt{R}_1, \texttt{L}_2, \texttt{R}_2, \texttt{BWT}_1, \texttt{BWT}_2))$ ▷ New intervals
  else
    $c_1^1, \ldots, c_{k_1}^1 \leftarrow \texttt{BWT}_1\texttt{.rangeDistinct}(\texttt{L}_1, \texttt{R}_1)$
    $c_1^2, \ldots, c_{k_2}^2 \leftarrow \texttt{BWT}_2\texttt{.rangeDistinct}(\texttt{L}_2, \texttt{R}_2)$
    $\{c_1 \ldots c_k\} \leftarrow \{c_1^1, \ldots, c_{k_1}^1\} \cup \{c_1^2, \ldots, c_{k_2}^2\}$
    for $i = 1 \ldots k$ **do**
      $\langle L_1^i, R_1^i \rangle \leftarrow \texttt{BWT}_1\texttt{.bwsearch}(\langle \texttt{L}_1, \texttt{R}_1 \rangle, \texttt{c}_\texttt{i})$ ▷ Backward search step
    end for
    for $i = 1 \ldots k$ **do**
      $\langle L_2^i, R_2^i \rangle \leftarrow \texttt{BWT}_2\texttt{.bwsearch}(\langle \texttt{L}_2, \texttt{R}_2 \rangle, \texttt{c}_\texttt{i})$ ▷ Backward search step
    end for
    $\langle \hat{L}_1^i, \hat{R}_1^i, \hat{L}_2^i, \hat{R}_2^i, \rangle_{i=1,\ldots,k} \leftarrow \texttt{sort}(\langle L_1^i, R_1^i, L_2^i, R_2^i, \rangle_{i=1,\ldots,k})$
    for $i = k \ldots 1$ **do**
      $\texttt{P.push}(\hat{L}_1^i, \hat{R}_1^i, \hat{L}_2^i, \hat{R}_2^i)$ ▷ Push in order of decreasing length
    end for
  end if
end while

index	BWT($\mathcal{S}_1$)	Suffixes
1	T	#
2	T	#
3	#	A A T #
4	A	A T #
5	G	C T #
6	#	G C T #
7	C	T #
8	A	T #

index	BWT($\mathcal{S}_2$)	Suffixes
1	T	#
2	T	G T #
3	G	T #
4	#	T G T #

index	DA	BWT($\mathcal{S}_1 \cup \mathcal{S}_2$)	Suffixes
1	0	T	#
2	0	T	#
3	1	T	#
4	0	#	A A T #
5	0	A	A T #
6	0	G	C T #
7	0	#	G C T #
8	1	T	G T #
9	0	C	T #
10	0	A	T #
11	1	G	T #
12	1	#	T G T #

Figure 4: Running example for Algorithm 3, merging the BWTs of two sets of strings. **Top Left**: BWT of the set $\mathcal{S}_1 = \{GCT\#, AAT\#\}$. **Top Right**: BWT of the set $\mathcal{S}_2 = \{TGT\#\}$. **Bottom**: BWT of the (ordered) union $\mathcal{S}_1 \cup \mathcal{S}_2 = \{GCT\#, AAT\#, TGT\#\}$. We use the colors black/red to show suffixes and BWT characters coming from the sets $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively. The document array (DA), computed by Algorithm 3, encodes these numbers (0 for black and 1 for red) and thus is sufficient to specify how the characters from the two input BWTs are interleaved in the output. The algorithm starts with the ranges of # on the two BWTs: $\langle L_1, R_1, L_2, R_2 \rangle = \langle 1, 2, 1, 1 \rangle$. In Lines 9-14, we set $DA[1, \ldots, 2] \leftarrow 0$ and $DA[3, \ldots, 3] \leftarrow 1$. These ranges can be left-extended only by letter T, yielding the ranges of $T\#$ on the two BWTs: $\langle L_1, R_1, L_2, R_2 \rangle = \langle 7, 8, 3, 3 \rangle$. This quadruple leads to the assignments $DA[9, \ldots, 10] \leftarrow 0$ and $DA[11, \ldots, 11] \leftarrow 1$ in Lines 9-14. By left-extending $T\#$ with A, we can now see an example of the case where one of the two BWT ranges becomes empty. By applying the LF mapping to these ranges with letter A, we obtain that the ranges of $AT\#$ on the two BWTs are: $\langle L_1, R_1, L_2, R_2 \rangle = \langle 4, 4, 2, 1 \rangle$ (note the empty range on $BWT(\mathcal{S}_2)$). The rule in lines 9-14 of the algorithm is still correct: we set $DA[5, \ldots, 5] \leftarrow 0$ and $DA[6, \ldots, 5] \leftarrow 1$. Note that the latter assignment is on an empty interval of DA, therefore the operation does not modify any bit of the array.

with just one cache miss. This is crucial for our algorithms, since at each step we need to left-extend ranges by all characters. This structure divides the text in blocks of 128 characters. Each block is stored using 512 cache-aligned bits (the typical size of a cache line), divided as follows. The first 128 bits store four 32-bits counters with the partial ranks of A, C, G, and T before the block (if the string is longer than 2^{32} characters, we further break it into superblocks of 2^{32} characters; on reasonably-large inputs, the extra rank table fits in cache and does not cause additional cache misses). The following three blocks of 128 bits store the first, second, and third bits, respectively, of the characters' binary encodings (each character is packed in 3 bits). Using this layout, the rank of each character in the block can be computed with at most three masks, a bitwise AND (actually less, since we always compute the rank of all five characters and we re-use partial results whenever possible), and a `popcount` operation. We also implemented a packed string on the augmented alphabet $\Sigma^+_{DNA} = \{A, C, G, N, T, \#\}$ using 4.38 bits per character and offering the same cache-efficiency guarantees. In this case, a 512-bits block stores 117 characters, packed as follows. As seen above, the first 128 bits store four 32-bits counters with the partial ranks of A, C, G, and T before the block. Each of the following three blocks of 128 bits is divided in a first part of 117 bits and a second part of 11 bits. The first parts store the first, second, and third bits, respectively, of the characters' binary encodings. The three parts of 11 bits, concatenated together, store the rank of N's before the block. This layout minimizes the number of bitwise operations (in particular, shifts and masks) needed to compute a parallel rank.

Several heuristics have been implemented to reduce the number of cache misses in practice. In particular, we note that in Algorithm 2 we can avoid backtracking when the range size becomes equal to one; the same optimization can be implemented in Algorithm 3 when also computing the LCP array, since leaves of size one can be identified during navigation of internal suffix tree nodes. Overall, we observed (using a memory profiler) that in practice the combination of Algorithms 1-2 generates at most $1.5n$ cache misses, n being the total collection's size. The extension of Algorithm 3 that computes also LCP values generates twice this number of cache misses (this is expected, since the algorithm navigates two BWTs).

We now report some preliminary experiments on our algorithms: `bwt2lcp` (Algorithms 1-2) and `merge` (Algorithm 3, extended to compute also the LCP array). All tests were done on a DELL PowerEdge R630 machine, used in non exclusive mode. Our platform is a 24-core machine with Intel(R) Xeon(R) CPU E5-2620 v3 at 2.40 GHz, with 128 GiB of shared memory and 1TB of SSD. The system is Ubuntu 14.04.2 LTS. The code was compiled using gcc 8.1.0 with flags `-Ofast -fstrict-aliasing`.

Table 1 summarizes the datasets used in our experiments. "NA12891.8"[2]

[2] `ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3/data/NA12891/sequence_read/SRR622458_1.filt.fastq.gz`

Name	Size GiB	σ	N. of reads	Max read length	bytes for lcp values
NA12891.8	8.16	5	85,899,345	100	1
shortreads	8.0	6	85,899,345	100	1
pacbio	8.0	6	942,248	71,561	4
pacbio.1000	8.0	6	8,589,934	1000	2
NA12891.24	23.75	6	250,000,000	100	1
NA12878.24	23.75	6	250,000,000	100	1

Table 1: Datasets used in our experiments. Size accounts only for the alphabet's characters. The alphabet's size σ includes the terminator.

	Preprocessing		`eGap`		`merge`	
Name	Wall Clock (h:mm:ss)	RAM (GiB)	Wall Clock (h:mm:ss)	RAM (GiB)	Wall Clock (h:mm:ss)	RAM (GiB)
NA12891.8	1:15:57	2.84	10:15:07	18.09 (-m 32000)	3:16:40	26.52
NA12891.8.RC	1:17:55	2.84				
shortreads	1:14:51	2.84	11:03:10	16.24 (-m 29000)	3:36:21	26.75
shortreads.RC	1:19:30	2.84				
pacbio.1000	2:08:56	31.28	5:03:01	21.23 (-m 45000)	4:03:07	42.75
pacbio.1000.RC	2:15:08	31.28				
pacbio	2:27:08	31.25	2:56:31	33.40 (-m 80000)	4:38:27	74.76
pacbio.RC	2:19:27	31.25				
NA12878.24	4:24:27	7.69	31:12:28	47.50 (-m 84000)	6:41:35	73.48
NA12891.24	4:02:42	7.69				

Table 2: In this experiment, we merge pairs of BWTs and induce the LCP of their union using `eGap` and `merge`. We also show the resources used by the pre-processing step (building the BWTs) for comparison. Wall clock is the elapsed time from start to completion of the instance, while RAM (in GiB) is the peak Resident Set Size (RSS). All values were taken using the `/usr/bin/time` command. During the preprocessing step on the collections pacBio.1000 and pacBio, the available memory in MB (parameter m) of `eGap` was set to 32000 MB. In the merge step this parameter was set to about to the memory used by `merge`. `eGap` and `merge` take as input the same BWT file.

contains Human DNA reads on the alphabet Σ_{DNA} where we have removed reads containing the nucleotide N. "shortreads" contains Human DNA short reads on the extended alphabet Σ^+_{DNA}. "pacbio" contains PacBio RS II reads from the species *Triticum aestivum* (wheat). "pacbio.1000" are the strings from "pacbio" trimmed to length 1,000. All the above datasets except the first have been download from `https://github.com/felipelouza/egap/tree/master/dataset`. To conclude, we added two collections, "NA12891.24" and "NA12878.24" obtained by taking the first $250,000,000$ reads from individuals NA12878[3] and NA12891. All datasets except "NA12891.8" are on the alphabet Σ^+_{DNA}. In Tables 2 and 3, the suffix ".RC" added to a dataset's name indicates the reverse-complemented dataset.

We compare our algorithms with `eGap`[4] and BCR[5], two tools designed to build the BWT and LCP of a set of DNA reads. Since no tools for inducing the LCP from the BWT of a set of strings are available in the literature, in Table

[3] `ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3/data/NA12878/sequence_read/SRR622457_1.filt.fastq.gz`

[4] `https://github.com/felipelouza/egap`

[5] `https://github.com/giovannarosone/BCR_LCP_GSA`

	Preprocessing		bwt2lcp	
Name	Wall Clock (h:mm:ss)	RAM GiB	Wall Clock (h:mm:ss)	RAM (GiB)
NA12891.8 ∪ NA12891.8.RC (BCR)	2:43:02	5.67	1:40:01	24.48
shortread ∪ shortread.RC (BCR)	2:47:07	5.67	2:14:41	24.75
pacbio.1000 ∪ pacbio.1000.RC (eGap-m 32000)	7:07:46	31.28	1:54:56	40.75
pacbio ∪ pacbio.RC (eGap-m 80000)	6:02:37	78.125	2:14:37	72.76
NA12878.24 ∪ NA12891.24 (BCR)	8:26:34	16.63	6:41:35	73.48

Table 3: In this experiment, we induced the LCP array from the BWT of a collection (each collection is the union of two collections from Table 2). We also show pre-processing requirements (i.e. building the BWT) of the better performing tool between BCR and eGap.

3 we simply compare the resources used by bwt2lcp with the time and space requirements of eGap and BCR when building the BWT. In [23], experimental results show that BCR works better on short reads and collections with a large average LCP, while eGap works better when the datasets contain long reads and relatively small average LCP. For this reason, in the preprocessing step we have used BCR for the collections containing short reads and eGap for the other collections. eGap, in addition, is capable of merging two or more BWTs while inducing the LCP of their union. In this case, we can therefore directly compare the performance of eGap with our tool merge; results are reported in Table 2. Since the available RAM is greater than the size of the input, we have used the semi-external strategy of eGap. Notice that an entirely like-for-like comparison between our tools and eGap is not completely feasible, since eGap is a semi-external memory tool (our tools, instead, use internal memory only). While in our tables we report RAM usage only, it is worth noticing that eGap uses a considerable amount of disk working space. For example, the tool uses 56GiB of disk working space when run on a 8GiB input (in general, the disk usage is of $7n$ bytes).

Our tools exhibit a dataset-independent linear time complexity, whereas eGap's running time depends on the average LCP. Table 3 shows that our tool bwt2lcp induces the LCP from the BWT faster than building the BWT itself. When N's are not present in the dataset, bwt2lcp processes data at a rate of 2.92 megabases per second and uses 0.5 bytes per base in RAM in addition to the LCP. When N's are present, the throughput decreases to 2.12 megabases per second and the tool uses 0.55 bytes per base in addition to the LCP. As shown in Table 2, our tool merge is from 1.25 to 4.5 times faster than eGap on inputs with large average LCP, but 1.6 times slower when the average LCP is small (dataset "pacbio"). When N's are not present in the dataset, merge processes data at a rate of 1.48 megabases per second and uses 0.625 bytes per base in addition to the LCP. When N's are present, the throughput ranges from 1.03 to 1.32 megabases per second and the tool uses 0.673 bytes per base in addition to the LCP. When only computing the merged BWT (results not shown here for space reasons), merge uses in total 0.625/0.673 bytes per base in RAM (without/with N's) and is about 1.2 times faster than the version computing also the LCP.

11. Acknowledgements

GR is partially, and NP is totally, supported by the project MIUR-SIR CMACBioSeq ("Combinatorial methods for analysis and compression of biological sequences") grant n. RBSI146R5L.

[1] M. Burrows, D. Wheeler, A Block Sorting data Compression Algorithm, Tech. rep., DEC Systems Research Center (1994).

[2] S. Mantaci, A. Restivo, G. Rosone, M. Sciortino, An extension of the Burrows-Wheeler Transform, Theor. Comput. Sci. 387 (3) (2007) 298–312.

[3] M. Bauer, A. Cox, G. Rosone, Lightweight algorithms for constructing and inverting the BWT of string collections, Theor. Comput. Sci. 483 (0) (2013) 134–148.

[4] J. Kärkkäinen, Fast BWT in small space by blockwise suffix sorting, Theor. Comput. Sci. 387 (3) (2007) 249–257. doi:10.1016/j.tcs.2007.07.018.

[5] A. Policriti, N. Gigante, N. Prezza, Average Linear Time and Compressed Space Construction of the Burrows-Wheeler Transform, in: Language and Automata Theory and Applications, Springer International Publishing, Cham, 2015, pp. 587–598.

[6] T. Beller, M. Zwerger, S. Gog, E. Ohlebusch, Space-Efficient Construction of the Burrows-Wheeler Transform, in: String Processing and Information Retrieval, Springer International Publishing, Cham, 2013, pp. 5–16.

[7] J. Fuentes-Seplveda, G. Navarro, Y. Nekrich, Space-efficient computation of the burrows-wheeler transform, in: 2019 Data Compression Conference (DCC), 2019, pp. 132–141. doi:10.1109/DCC.2019.00021.

[8] D. Kempa, T. Kociumaka, String Synchronizing Sets: Sublinear-time BWT Construction and Optimal LCE Data Structure, in: Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, ACM, New York, NY, USA, 2019, pp. 756–767. doi:10.1145/3313276.3316368.

[9] G. Navarro, Y. Nekrich, Optimal dynamic sequence representations, SIAM Journal on Computing 43 (5) (2014) 1781–1806.

[10] A. Cox, F. Garofalo, G. Rosone, M. Sciortino, Lightweight LCP construction for very large collections of strings, J. Discrete Algorithms 37 (2016) 17–33.

[11] N. Prezza, N. Pisanti, M. Sciortino, G. Rosone, Detecting Mutations by eBWT, in: WABI 2018, Vol. 113 of LIPIcs, 2018, pp. 3:1–3:15.

[12] N. Prezza, N. Pisanti, M. Sciortino, G. Rosone, SNPs detection by eBWT positional clustering, Algorithms Mol. Biol. 14 (1) (2019) 3.

[13] V. Guerrini, G. Rosone, Lightweight Metagenomic Classification via eBWT, in: Algorithms for Computational Biology, Vol. 11488 LNBI, Springer International Publishing, 2019, pp. 112–124.

[14] W. Tustumi, S. Gog, G. Telles, F. Louza, An improved algorithm for the all-pairs suffix-prefix problem, Journal of Discrete Algorithms 37 (2016) 34 – 43. doi:https://doi.org/10.1016/j.jda.2016.04.002.

[15] K. Sadakane, Succinct representations of lcp information and improvements in the compressed suffix arrays, in: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '02, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002, pp. 225–232.

[16] T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, in: Combinatorial Pattern Matching, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, pp. 181–192.

[17] R. Grossi, J. S. Vitter, Compressed suffix arrays and suffix trees with applications to text indexing and string matching, SIAM J. Comput. 35 (2) (2005) 378–407. doi:10.1137/S0097539702402354.

[18] K. Sadakane, Compressed suffix trees with full functionality, Theor. Comp. Sys. 41 (4) (2007) 589–607. doi:10.1007/s00224-006-1198-x.

[19] E. Ohlebusch, J. Fischer, S. Gog, CST++, in: String Processing and Information Retrieval, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 322–333.

[20] J. Holt, L. McMillan, Constructing Burrows-Wheeler transforms of large string collections via merging, in: ACM-BCB, ACM, 2014, pp. 464–471.

[21] J. Holt, L. McMillan, Merging of multi-string BWTs with applications, Bioinformatics 30 (24) (2014) 3524–3531.

[22] P. Bonizzoni, G. Della Vedova, S. Nicosia, Y. Pirola, M. Previtali, R. Rizzi, Divide and conquer computation of the multi-string BWT and LCP array, in: CiE, LNCS, Springer, 2018, pp. 107–117.

[23] L. Egidi, F. Louza, G. Manzini, G. Telles, External memory BWT and LCP computation for sequence collections with applications, Algorithms Mol. Biol. 14 (1) (2019) 6.

[24] D. Belazzougui, Linear time construction of compressed text indices in compact space, in: Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing, STOC '14, ACM, New York, NY, USA, 2014, pp. 148–193. doi:10.1145/2591796.2591885.

[25] J. Kärkkäinen, G. Manzini, S. J. Puglisi, Permuted longest-common-prefix array, in: Combinatorial Pattern Matching, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 181–192.

[26] N. Välimäki, V. Mäkinen, W. Gerlach, K. Dixit, Engineering a compressed suffix tree implementation, J. Exp. Algorithmics 14 (2010) 2:4.2–2:4.23. doi:10.1145/1498698.1594228.

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

[28] J. I. Munro, G. Navarro, Y. Nekrich, Space-efficient construction of compressed indexes in deterministic linear time, in: SODA, SIAM, 2017, pp. 408–424.

[29] J. I. Munro, V. Raman, Succinct representation of balanced parentheses, static trees and planar graphs, in: Proceedings of the 38th Annual Symposium on Foundations of Computer Science, FOCS '97, IEEE Computer Society, Washington, DC, USA, 1997, pp. 118–.

[30] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, S. S. Rao, Representing trees of higher degree, Algorithmica 43 (4) (2005) 275–292. doi:10.1007/s00453-004-1146-6.

[31] W.-K. Hon, K. Sadakane, W.-K. Sung, Breaking a time-and-space barrier in constructing full-text indices, SIAM J. Comput. 38 (6) (2009) 2162–2178. doi:10.1137/070685373.

[32] D. Belazzougui, F. Cunial, J. Kärkkäinen, V. Mäkinen, Linear-time string indexing and analysis in small space, arXiv preprint arXiv:1609.06378.

[33] N. Prezza, G. Rosone, Space-Efficient Computation of the LCP Array from the Burrows-Wheeler Transform, in: 30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019), Vol. 128 of Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2019, pp. 7:1–7:18. doi:10.4230/LIPIcs.CPM.2019.7.

[34] S. Puglisi, A. Turpin, Space-time tradeoffs for longest-common-prefix array computation, in: ISAAC, Vol. 5369 of LNCS, Springer, 2008, pp. 124–135.

[35] F. Shi, Suffix arrays for multiple strings: A method for on-line multiple string searches, in: ASIAN, Vol. 1179 of LNCS, Springer, 1996, pp. 11–22.

[36] F. Louza, G. Telles, S. Hoffmann, C. Ciferri, Generalized enhanced suffix array construction in external memory, Algorithms Mol. Biol. 12 (1) (2017) 26.

[37] U. Manber, G. Myers, Suffix arrays: A new method for on-line string searches, SIAM Journal on Computing 22 (5) (1993) 935–948. doi:10.1137/0222058.

[38] G. Navarro, Wavelet trees for all, J. Discrete Algorithms 25 (2014) 2 – 20.

[39] P. Ferragina, G. Manzini, Opportunistic data structures with applications, in: FOCS, IEEE, 2000, pp. 390–398.

[40] F. Claude, G. Navarro, A. Ordónez, The wavelet matrix: An efficient wavelet tree for large alphabets, Information Systems 47 (2015) 15–32.

[41] D. Belazzougui, G. Navarro, Alphabet-independent compressed text indexing, TALG 10 (4) (2014) 23.

[42] G. Manzini, Two Space Saving Tricks for Linear Time LCP Array Computation, in: T. Hagerup, J. Katajainen (Eds.), Algorithm Theory - SWAT 2004, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 372–383.

[43] G. Navarro, Compact Data Structures: A Practical Approach, 1st Edition, Cambridge University Press, New York, NY, USA, 2016.

[44] L. Egidi, G. Manzini, Lightweight BWT and LCP merging via the Gap algorithm, in: SPIRE, LNCS, Springer, 2017, pp. 176–190.