Space-Efficient Computation of the LCP Array from the Burrows-Wheeler Transform

Nicola Prezza
Department of Computer Science, University of Pisa, Italy
nicola.prezza@di.unipi.it

Giovanna Rosone
Department of Computer Science, University of Pisa, Italy
giovanna.rosone@unipi.it

Abstract

We show that the Longest Common Prefix Array of a text collection of total size n on alphabet $[1, \sigma]$ can be computed from the Burrows-Wheeler transformed collection in $O(n \log \sigma)$ time using $o(n \log \sigma)$ bits of working space on top of the input and output. Our result improves (on small alphabets) and generalizes (to string collections) the previous solution from Beller et al., which required $O(n)$ bits of extra working space. We also show how to merge the BWTs of two collections of total size n within the same time and space bounds. The procedure at the core of our algorithms can be used to enumerate suffix tree intervals in succinct space from the BWT, which is of independent interest. An engineered implementation of our first algorithm on DNA alphabet induces the LCP of a large (16 GiB) collection of short (100 bases) reads at a rate of 2.92 megabases per second using in total 1.5 Bytes per base in RAM. Our second algorithm merges the BWTs of two short-reads collections of 8 GiB each at a rate of 1.7 megabases per second and uses 0.625 Bytes per base in RAM. An extension of this algorithm that computes also the LCP array of the merged collection processes the data at a rate of 1.48 megabases per second and uses 1.625 Bytes per base in RAM.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Theory of computation → Data compression

Keywords and phrases Burrows-Wheeler Transform, LCP array, DNA reads

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.7

Related Version A full version of the paper is available at `https://arxiv.org/abs/1901.05226`.

Funding 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 Introduction

The increasingly-growing production of huge datasets composed of short strings – 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 [7] and its extension to sets of strings [16, 1] is becoming the gold-standard in the field: even when not compressed, its size is one order of magnitude smaller than classic suffix arrays (while preserving many of their indexing capabilities). The functionalities of this transformation can be extended by computing additional structures such as the LCP array [9]; see, e.g. [19, 20] for a bioinformatics application where this component is needed. To date, several practical algorithms have been developed to solve the task of merging or building *de novo* such components [1, 9, 13, 14, 6, 10], but little work has been devoted to the task of computing the LCP array from the BWT of string collections in little space (internal and external working space). The only existing work we are aware of in this direction is from Beller et al. [5], who show how to build the LCP array from the BWT of a single text in $O(n \log \sigma)$

time and $O(n)$ bits of working space on top of the input and output. Other works [17, 2] show how to build the LCP array directly from the text in $O(n)$ time and $O(n \log \sigma)$ bits of space (compact). In this paper, we combine Beller et al.'s algorithm with a recent suffix-tree enumeration procedure of Belazzougui [2] and reduce this working space to $o(n \log \sigma)$ while also generalizing the algorithm to string collections. As a by-product, we show an algorithm able to merge the BWTs of two string collections using just $o(n \log \sigma)$ bits of working space. An efficient implementation of our algorithms on DNA alphabet uses (in RAM) as few as n bits on top of a packed representation of the input/output, and can process data as fast as 2.92 megabases per second.

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 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 $1 \leq i \leq j \leq n$. 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.

Let $\mathcal{C} = \{T_1, \ldots, T_m\}$ 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 $\mathcal{C}[i] = T_i$.

The *generalized suffix array* $GSA[1..n]$ (see [21, 9, 15]) of $\mathcal{C}$ is an array of pairs $GSA[i] = \langle j, k \rangle$ such that $\mathcal{C}[j][k..]$ is the i-th lexicographically smallest suffix of strings in $\mathcal{C}$, where we break ties by input position (i.e. j in the notation above). We denote by $\texttt{range}(\texttt{W}) = \langle \texttt{left}(\texttt{W}), \texttt{right}(\texttt{W}) \rangle$ the maximal pair $\langle L, R \rangle$ such that all suffixes in $GSA[L, R]$ are prefixed by W. 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.

The *extended Burrows-Wheeler Transform* $BWT[1..n]$ [16, 1] of $\mathcal{C}$ is the character array defined as $BWT[i] = \mathcal{C}[j][k - 1 \bmod |\mathcal{C}[j]|]$, where $\langle j, k \rangle = GSA[i]$.

The *longest common prefix* (LCP) array of a collection $\mathcal{C}$ of strings (see [9, 15, 10]) is an array storing the length of the longest common prefixes between two consecutive suffixes of $\mathcal{C}$ in lexicographic order (with $LCP[1] = 0$).

Given two collections $\mathcal{C}_1, \mathcal{C}_2$ 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 $\mathcal{C}_1$. When merging suffixes of the two collections, ties are broken by collection number (i.e. suffixes of $\mathcal{C}_1$ are smaller than suffixes of $\mathcal{C}_2$ 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). Alternatively, $C[c]$ 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 [18]. $S.rank_c(i)$ is the number of characters equal to c in $S[1, i - 1]$.

Function $\texttt{getIntervals}(\texttt{L}, \texttt{R}, \texttt{BWT})$, where BWT is the extended Burrows-Wheeler transform of a string collection and $\langle L, R \rangle$ is the suffix array interval of some string W appearing in the collection, returns all suffix array intervals of strings cW, with $c \neq \#$, that occur in the collection. 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 [5]. 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 [5] (this requires to DFS-visit the sub-tree of the wavelet tree induced by $BWT[L, R]$; the visit requires only $\log \sigma$ bits to store the current path in the tree).

We note that 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.

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 procedure that, given the suffix array interval $\langle L, R \rangle$ of a string W, returns the suffix array interval of cW by using the BWT [12]. 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.

3 Our Contributions

Our work builds upon the following two results from Belazzougui[1] [2] and Beller et al. [5].

▶ **Theorem 1** (Belazzougui [2]). *Given the Burrows-Wheeler Transform of a text $T \in [1, \sigma]^n$ represented with a wavelet tree, we can solve the following problem in $O(n \log \sigma)$ time using $O(\sigma^2 \log^2 n)$ bits of working space on top of the BWT. 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.*

▶ **Theorem 2** (Beller et al. [5]). *Given the Burrows-Wheeler Transform of a text T represented with a wavelet tree, we can compute the LCP array of T in $O(n \log \sigma)$ time using $4n$ bits of working space on top of the BWT and the LCP.*

Theorem 2 represents the state of the art for computing the LCP array from the BWT. Our first observation is that Theorem 1 can be directly used to induce the LCP array of T using just $O(\sigma^2 \log^2 n)$ bits of working space on top of the input and output (proof in Section 6.1). In Section 6.1 we combine this result with Theorem 2 and obtain our first theorem:

▶ **Theorem 3.** *Given the Burrows-Wheeler Transform of a text $T \in [1, \sigma]^n$, we can compute the LCP array of T in $O(n \log \sigma)$ time using $o(n \log \sigma)$ bits of working space on top of the BWT and the LCP.*

Proof. First, we replace T by its wavelet matrix [8] – of size $n \log \sigma + o(n \log \sigma)$ bits – in $O(n \log \sigma)$ time using just n bits of additional working space as shown in [8]. Wavelet matrices support the same set of operations of wavelet trees in the same running times (indeed, they

[1] While the original theorem [2, Sec. 5.1] is general and uses the underlying rank data structure as a black box, in our case we strive for succinct space (not compact as in [2]) and stick to wavelet trees. All details on how to achieve the claimed running time and space are described in Section 4.

can be considered as a wavelet tree representation). We re-use a portion (n bits) of the LCP array's space (which always takes $\geq n$ bits) to accommodate the extra n bits required for building the wavelet matrix, so the overall working space does not exceed $o(n \log \sigma)$ bits on top of the BWT and LCP. In the rest of the paper we will simply assume that the input is represented by a wavelet tree.

At this point, 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 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 (not good since we aim at $o(n \log \sigma)$). ◀

We proceed by extending Theorem 1 to enumerate also the intervals corresponding to leaves of the generalized suffix tree of a collection (Theorem 1 enumerates internal nodes). In Section 6.1 we show that this simple modification, combined again with the strategy of Theorem 2 (generalized to collections), can be used to extend Theorem 3 to text collections:

▶ **Theorem 4.** *Given the Burrows-Wheeler Transform of a collection $\mathcal{C} = \{T_1, \ldots, T_m\}$ of total length n on alphabet $[1, \sigma]$, we can compute the LCP array of $\mathcal{C}$ in $O(n \log \sigma)$ time using $o(n \log \sigma)$ bits of working space on top of the BWT and the LCP.*

In [2, 3], 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 $k \geq 1$ [3, Thm. 7]. We show that our strategy enables a more space-efficient algorithm for the task of merging BWTs of collections. The following theorem, proved in Section 6.2, 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.

▶ **Theorem 5.** *Given the Burrows-Wheeler Transforms of two collections $\mathcal{C}_1$ and $\mathcal{C}_2$ of total length n on alphabet $[1, \sigma]$, we can compute the Document Array of $\mathcal{C}_1 \cup \mathcal{C}_2$ 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.*

When $k = \log \sigma$, the running time of [3, Thm. 7] is the same as our Theorem 5 but the working space is higher: $n \log \sigma + O(n)$ bits. We also briefly discuss how to extend Theorem 5 to build the LCP array of the merged collection. In Section 7 we present an implementation of our algorithms and an experimental comparison with `eGap` [11], the state-of-the-art tool designed for the same task of merging BWTs while inducing the LCP of their union.

4 Belazzougui's Enumeration Algorithm

In [2], 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. In this section we describe his algorithm.

Remember that explicit suffix tree nodes correspond to right-maximal text substrings. By definition, for any right-maximal substring W there exist at least two distinct characters $c_1, \ldots, c_k$ such that Wc_i is a substring of T, for $i = 1, \ldots, k$. The first idea is to represent any text substring W (not necessarily right-maximal) as follows. Let $\texttt{chars}_{\texttt{W}}[1] < \cdots < \texttt{chars}_{\texttt{W}}[\texttt{k}_{\texttt{W}}]$

be the (lexicographically-sorted) character array such that $W \cdot \texttt{chars}_\texttt{W}[\texttt{i}]$ is a substring of T for all[2] $i = 1, \ldots, k_W$, where k_W is the number of right-extensions of W. Let moreover $\texttt{first}_\texttt{W}[\texttt{1..k}_\texttt{W} + \texttt{1}]$ 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 [2], we omit $\texttt{chars}_\texttt{W}$ from the representation and we add $|W|$; these modifications will turn useful later): $\texttt{repr}(\texttt{W}) = \langle \texttt{first}_\texttt{W}, |\texttt{W}| \rangle$. Note that, if W is not right-maximal and is not 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). At this point, the enumeration algorithm works by visiting the Weiner Link tree of T starting from the root's representation $\texttt{repr}(\epsilon) = \langle \texttt{first}_\epsilon, \texttt{0} \rangle$, 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 all (and only) the sorted alphabet's characters. 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. If nodes are pushed on the stack in decreasing order of interval length, then the stack's size never exceeds $O(\sigma \log n)$. In Appendix A we describe in detail how Weiner links are computed, and show that with this strategy we visit all suffix tree nodes in $O(n \log \sigma)$ time using overall $O(\sigma^2 \log^2 n)$ bits of additional space (for the stack). In Section 6.1 we show that this enumeration algorithm can be used to compute the LCP array from the BWT of a collection.

5 Beller et al.'s Algorithm

Also Beller et al.'s algorithm [5] 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 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 $\texttt{range}(\texttt{W})$ 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 size of Belazzougui's stack can be upper-bounded by $O(\sigma \log n)$, but its elements take more space to be represented.

Beller et al.'s algorithm starts by initializing all LCP entries to $\perp$ (an undefined value), and 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 $\texttt{getIntervals}(\texttt{L}, \texttt{R}, \texttt{BWT})$ [5] (see Section 2) we left-extend the BWT interval $\langle L, R \rangle$ with the characters $c_1, \ldots, c_k$ in $\texttt{rangeDistinct}(\texttt{L}, \texttt{R})$, 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 $LCP[R_i + 1] = \perp$ then we set $LCP[R_i + 1] \leftarrow \ell$ 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

[2] We require $\texttt{chars}_\texttt{W}$ to be also complete: if Wc is a substring of T, then $c \in \texttt{chars}_\texttt{W}$.

Algorithm 1: `Node-Type(BWT,LCP)`.

input : Wavelet tree of the Burrows-Wheeler transformed collection `BWT` $\in [1,\sigma]^n$ and empty array `LCP[1..n]`.
behavior : Fills node-type `LCP` values.

```
1  if σ > √n / log² n then
2  |  BGOS(BWT, LCP);                                   /* 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 |  |  |  LCP[first_W[i]] ← ℓ;                      /* Set LCP value */
11 |  |  x_1, ..., x_k ← BWT.Weiner(⟨first_W, ℓ⟩);    /* Follow Weiner Links */
12 |  |  x'_1, ..., x'_k ← sort(x_1, ..., x_k);       /* Sort by interval length */
13 |  |  for i = k ... 1 do
14 |  |  |  P.push(x'_i);                             /* Push representations */
15 LCP[0] ← 0;
```

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). To limit space usage, Beller et al. use two different queue representations. As long as there are $O(n/\log n)$ elements in the queue, they use a simple vector. When there are more intervals, they switch to a representation based on four bitvectors of length n that still guarantees constant amortized operations. All details are described in Appendix B. Beller et al. [5] show that the above algorithm correctly computes the LCP array of a text. In the next section we generalize the algorithm to text collections.

6 Our Algorithms

We describe our algorithms directly on string collections. This will include, as a particular case, inputs formed by a single text. Procedure `BGOS(BWT,LCP)` in Line 2 of Algorithm 1 is a call to Beller et al.'s algorithm, modified as follows. First, we set `LCP[C[c]]` $\leftarrow 0$ for all $c \in \Sigma$. Then, we push in the queue $\langle$`range(c)`$, 1\rangle$ 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 #.

6.1 Computing the LCP From the BWT

Let $\mathcal{C}$ be a text collection where each string is ended by a terminator # (common to all strings). Consider now the LCP and GSA (generalized Suffix Array) arrays of $\mathcal{C}$. We divide LCP values in two types. Let $GSA[i] = \langle j,k\rangle$, with $i > 1$, indicate that the i-th suffix in the lexicographic ordering of all suffixes of strings in $\mathcal{C}$ is $\mathcal{C}[j][k..]$. A LCP value `LCP[i]` is of *node type* when the i-th and $(i-1)$-th suffixes are distinct: $\mathcal{C}[j][k..] \neq \mathcal{C}[j'][k'..]$, 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 $\mathcal{C}[j][k..k+LCP[i]-1]$). Note that it might be that one of the two suffixes, $\mathcal{C}[j][k..]$ or $\mathcal{C}[j'][k'..]$, is the empty string (followed by the terminator) #. Similarly, a *leaf-type* LCP value `LCP[i]` is such that the i-th and $(i-1)$-th suffixes are equal: $\mathcal{C}[j][k..] = \mathcal{C}[j'][k'..]$. 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, $\mathcal{C}[j][k..] = \mathcal{C}[j'][k'..]$ could coincide with #. Entry $LCP[0]$ escapes the above classification, so we will set 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 11 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 Appendix A. 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. 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.[5] and described in Appendix B. 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 13-16 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 initialize an empty array `LCP[1..n]`, (2), we fill node-type entries using procedure `Node-Type(BWT, LCP)` described in Algorithm 1, and (3) we fill leaf-type entries using procedure `Leaf-Type(BWT, LCP)` described in Algorithm 2.

Algorithm 2: `Leaf-Type(BWT,LCP)`.

input : Wavelet tree of the Burrows-Wheeler transformed collection `BWT` $\in [1, \sigma]^n$ and array `LCP[1..n]`.
behavior : Fills leaf-type `LCP` values.

```
1  if σ > n/log³ n then
2  |  P ← new_queue();                                   /* Initialize new queue */
3  else
4  |  P ← new_stack();                                   /* Initialize new stack */
5  P.push(BWT.range(#), 0);                  /* Push range of terminator and LCP value 0 */
6  while not P.empty() do
7  |  ⟨⟨L, R⟩, ℓ⟩ ← P.pop();                            /* Pop highest-priority element */
8  |  for i = L + 1 ... R do
9  |  |  LCP[i] ← ℓ;                                   /* Set LCP inside range of ST leaf */
10 |  if σ > n/log³ n then
11 |  |  P.push(getIntervals(L, R, BWT), ℓ + 1);       /* Pairs ⟨interval, ℓ + 1⟩ */
12 |  else
13 |  |  ⟨L_i, R_i⟩_{i=1,...,k} ← getIntervals(L, R, BWT);
14 |  |  ⟨L'_i, R'_i⟩_{i=1,...,k} ← sort(⟨L_i, R_i⟩_{i=1,...,k});   /* Sort by interval length */
15 |  |  for i = k ... 1 do
16 |  |  |  P.push(⟨L'_i, R'_i⟩, ℓ + 1);              /* Push in order of decreasing length */
```

Theorems 3 and 4 follow from the correctness of our procedure, which for space reasons is reported in Appendix C as Lemma 6. As a by-product, in Appendix D we note that Algorithm 1 can be used to enumerate suffix tree intervals in succinct space from the BWT, which could be of independent interest.

6.2 Merging BWTs in Small Space

The procedure of Algorithm 2 can be extended to merge BWTs of two collections $\mathcal{C}_1$, $\mathcal{C}_2$ 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 $\mathcal{C}_1 \cup \mathcal{C}_2$ (note: for us, a collection is an ordered multi-set of strings). Our procedure differs from that described in [3, 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 4, 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 $BWT(\mathcal{C}_1)$ and $BWT(\mathcal{C}_2)$, of strings of the form $W\#$. Note that: (i) the suffix array of $\mathcal{C}_1 \cup \mathcal{C}_2$ 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 $GSA(\mathcal{C}_1 \cup \mathcal{C}_2)$ can be partitioned as $\langle L, M\rangle \cdot \langle M+1, R\rangle$, where $\langle L, M\rangle$ contains only suffixes from $\mathcal{C}_1$ and $\langle M+1, R\rangle$ contains only suffixes from $\mathcal{C}_2$ (one of these two intervals could be empty). It follows that we can navigate in parallel the leaves of the suffix trees of $\mathcal{C}_1$ and $\mathcal{C}_2$ (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 $BWT(\mathcal{C}_1 \cup \mathcal{C}_2)$ comes from $BWT(\mathcal{C}_1)$ ($DA[i] = 0$) or $BWT(\mathcal{C}_2)$ ($DA[i] = 1$). 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 $\mathcal{C}_1$ and $\mathcal{C}_2$, 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 $\mathcal{C}_j$ that are smaller than $W\#$. Then, in the collection $\mathcal{C}_1 \cup \mathcal{C}_2$ 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 $\mathcal{C}_1 \cup \mathcal{C}_2$ 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 $\mathcal{C}_1$. 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. 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 Appendix B with Beller et al.'s representation. Recall that, at any time, the queue can be partitioned in 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 $|BWT(\mathcal{C}_1)| = n_1$, $|BWT(\mathcal{C}_2)| = n_2$, and $|BWT(\mathcal{C}_1 \cup \mathcal{C}_2)| = n = n_1 + n_2$. We keep two

bitvectors $\texttt{Open}[1..n]$ and $\texttt{Close}[1..n]$ storing opening and closing parentheses of intervals in $BWT(\mathcal{C}_1 \cup \mathcal{C}_2)$. We moreover keep two bitvectors $\texttt{NonEmpty}_1[1..n]$ and $\texttt{NonEmpty}_2[1..n]$ keeping track, for each i such that $\texttt{Open}[i] = 1$, of whether the interval starting in $BWT(\mathcal{C}_1 \cup \mathcal{C}_2)[i]$ contains suffixes of reads coming from $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively. Finally, we keep four bitvectors $\texttt{Open}_j[1..n_j]$ and $\texttt{Close}_j[1..n_j]$, for $j = 1, 2$, storing non-empty intervals on $BWT(\mathcal{C}_1)$ and $BWT(\mathcal{C}_2)$, 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 $\texttt{Open}[L] \leftarrow 1$ and $\texttt{Close}[R] \leftarrow 1$. Then, for $j = 1, 2$, we set $\texttt{NonEmpty}_j[L] \leftarrow 1$, $\texttt{Open}_j[L_j] \leftarrow 1$ and $\texttt{Close}_j[R_j] \leftarrow 1$ if and only if $R_j \geq L_j$. This queue representation takes $O(n)$ bits. By construction, for each bit set in $\texttt{Open}$ at position i, there is a corresponding bit set in $\texttt{Open}_j$ if and only if $\texttt{NonEmpty}_j[i] = 1$ (moreover, corresponding bits set appear in the same order in $\texttt{Open}$ and $\texttt{Open}_j$). It follows that a left-to-right scan of these bitvectors is sufficient to identify corresponding intervals on $BWT(\mathcal{C}_1 \cup \mathcal{C}_2)$, $BWT(\mathcal{C}_1)$, and $BWT(\mathcal{C}_2)$. 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 [5]) by scanning in parallel the bitvectors forming the queue. Particular care needs to be taken only when we find the beginning of an interval $\texttt{Open}[L] = 1$ with $\texttt{NonEmpty}_1[L] = 0$ (the case $\texttt{NonEmpty}_2[L] = 0$ is symmetric). Let L_2 be the beginning of the corresponding non-empty interval on $BWT(\mathcal{C}_2)$. 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 $BWT(\mathcal{C}_1)$ 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 5. To conclude, we note that the algorithm can be 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), but for space reasons we do not describe all details here. 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 22, 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 $BWT(\mathcal{C}_1 \cup \mathcal{C}_2)$. Finally, note that Backward search can be performed correctly also when the input interval is empty: $\texttt{BWT}_j\texttt{.bwsearch}(\langle L_j, L_j - 1\rangle, c)$, where $L_j - 1$ is the number of suffixes in $\mathcal{C}_j$ smaller than some string W, correctly returns the pair $\langle L', R'\rangle$ such that L' is the number of suffixes in $\mathcal{C}_j$ 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.

7 Implementation and Experimental Evaluation

We implemented our algorithms on DNA alphabet in `https://github.com/nicolaprezza/bwt2lcp` using the language C++. Thanks 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 is more suited to 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

Algorithm 3: `Merge(BWT_1, BWT_2, DA)`.

input : Wavelet trees of the Burrows-Wheeler transformed collections $\texttt{BWT}_1 \in [1,\sigma]^{n_1}$, $\texttt{BWT}_2 \in [1,\sigma]^{n_2}$ and empty document array `DA[1..n]`, with $n = n_1 + n_2$.
behavior : Computes Document Array DA.

```
1  if σ > n/log^3 n then
2  |  P ← new_queue();                                  /* Initialize new queue of interval pairs */
3  else
4  |  P ← new_stack();                                  /* Initialize new stack of interval pairs */
5  P.push(BWT_1.range(#), BWT_2.range(#));              /* Push SA-ranges of terminator */
6  while not P.empty() do
7  |  ⟨L_1, R_1, L_2, R_2⟩ ← P.pop();                   /* Pop highest-priority element */
8  |  for i = L_1 + L_2 − 1 ... L_2 + R_1 − 1 do
9  |  |  DA[i] ← 0;                                    /* Suffixes from C_1 */
10 |  for i = L_2 + R_1 ... R_1 + R_2 do
11 |  |  DA[i] ← 1;                                    /* Suffixes from C_2 */
12 |  if σ > n/log^3 n then
13 |  |  P.push(getIntervals(L_1, R_1, L_2, R_2, BWT_1, BWT_2));   /* New intervals */
14 |  else
15 |  |  c^1_1, ..., c^1_{k_1} ← BWT_1.rangeDistinct(L_1, R_1);
16 |  |  c^2_1, ..., c^2_{k_2} ← BWT_2.rangeDistinct(L_2, R_2);
17 |  |  {c_1 ... c_k} ← {c^1_1, ..., c^1_{k_1}} ∪ {c^2_1, ..., c^2_{k_2}};
18 |  |  for i = 1 ... k do
19 |  |  |  ⟨L^i_1, R^i_1⟩ ← BWT_1.bwsearch(⟨L_1, R_1⟩, c_i);   /* Backward search step */
20 |  |  for i = 1 ... k do
21 |  |  |  ⟨L^i_2, R^i_2⟩ ← BWT_2.bwsearch(⟨L_2, R_2⟩, c_i);   /* Backward search step */
22 |  |  ⟨L̂^i_1, R̂^i_1, L̂^i_2, R̂^i_2, ⟩_{i=1,...,k} ← sort(⟨L^i_1, R^i_1, L^i_2, R^i_2, ⟩_{i=1,...,k});
23 |  |  for i = k ... 1 do
24 |  |  |  P.push(L̂^i_1, R̂^i_1, L̂^i_2, R̂^i_2);   /* Push in order of decreasing length */
```

compute the quintuple $\langle rank_c(i)\rangle_{i\in\Sigma_{DNA}}$ 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 class 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 before, 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

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

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 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.

	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				

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; the extension of Algorithm 3 that computes also LCP values generates twice this number of cache misses (this is expected, since it 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 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`.

	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 1 summarizes the datasets used in our experiments. "NA12891.8"[3] 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[4] 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`[5] and BCR[6], 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 we simply compare the resources used by `bwt2lcp` with the time and space requirements of `eGap` and BCR when building the BWT. In [10], 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, being `eGap` a semi-external memory tool (our tools, instead, use internal memory only). While in our tables we report RAM usage only, it is worth to notice 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,

[3] `ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3/data/NA12891/sequence_read/SRR622458_1.filt.fastq.gz`
[4] `ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3/data/NA12878/sequence_read/SRR622457_1.filt.fastq.gz`
[5] `https://github.com/felipelouza/egap`
[6] `https://github.com/giovannarosone/BCR_LCP_GSA`

`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.

References

1. M.J. Bauer, A.J. Cox, and G. Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theor. Comput. Sci.*, 483(0):134–148, 2013.
2. D. Belazzougui. Linear time construction of compressed text indices in compact space. In *STOC*, pages 148–193. ACM, 2014.
3. D. Belazzougui, F. Cunial, J. Kärkkäinen, and V. Mäkinen. Linear-time string indexing and analysis in small space. *arXiv preprint arXiv:1609.06378*, 2016.
4. D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. *TALG*, 10(4):23, 2014.
5. T. Beller, S. Gog, E. Ohlebusch, and T. Schnattinger. Computing the longest common prefix array based on the Burrows–Wheeler transform. *J. Discrete Algorithms*, 18:22–31, 2013.
6. P. Bonizzoni, G. Della Vedova, S. Nicosia, Y. Pirola, M. Previtali, and R. Rizzi. Divide and Conquer Computation of the Multi-string BWT and LCP Array. In *CiE*, LNCS, pages 107–117. Springer, 2018.
7. M. Burrows and D.J. Wheeler. A Block Sorting data Compression Algorithm. Technical report, DEC Systems Research Center, 1994.
8. F. Claude, G. Navarro, and A. Ordónez. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems*, 47:15–32, 2015.
9. A.J. Cox, F. Garofalo, G. Rosone, and M. Sciortino. Lightweight LCP construction for very large collections of strings. *J. Discrete Algorithms*, 37:17–33, 2016.
10. L. Egidi, F.A. Louza, G. Manzini, and G.P. Telles. External memory BWT and LCP computation for sequence collections with applications. *Algorithms Mol. Biol.*, 14(1):6, 2019.
11. L. Egidi and G. Manzini. Lightweight BWT and LCP merging via the Gap algorithm. In *SPIRE*, LNCS, pages 176–190. Springer, 2017.
12. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *FOCS*, pages 390–398. IEEE, 2000.
13. J. Holt and L. McMillan. Constructing Burrows-Wheeler transforms of large string collections via merging. In *ACM-BCB*, pages 464–471. ACM, 2014.
14. J. Holt and L. McMillan. Merging of multi-string BWTs with applications. *Bioinformatics*, 30(24):3524–3531, 2014.
15. F.A. Louza, G.P. Telles, S. Hoffmann, and C.D.A. Ciferri. Generalized enhanced suffix array construction in external memory. *Algorithms Mol. Biol.*, 12(1):26, 2017.
16. S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. An extension of the Burrows-Wheeler Transform. *Theor. Comput. Sci.*, 387(3):298–312, 2007.
17. J.I. Munro, G. Navarro, and Y. Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In *SODA*, pages 408–424. SIAM, 2017.
18. Gonzalo N. Wavelet trees for all. *J. Discrete Algorithms*, 25:2–20, 2014.
19. N. Prezza, N. Pisanti, M. Sciortino, and G. Rosone. Detecting Mutations by eBWT. In *WABI 2018*, volume 113 of *LIPIcs*, pages 3:1–3:15, 2018.

20 N. Prezza, N. Pisanti, M. Sciortino, and G. Rosone. SNPs detection by eBWT positional clustering. *Algorithms Mol. Biol.*, 14(1):3, 2019.

21 F. Shi. Suffix Arrays for Multiple Strings: A Method for On-Line Multiple String Searches. In *ASIAN*, volume 1179 of *LNCS*, pages 11–22. Springer, 1996.

A Notes on Belazzougui's Algorithm

The enumeration algorithm works by visiting the Weiner link tree of the text. While this guarantees that we will visit all and only the suffix tree's explicit nodes (see [2]), there are two main issues that need to be addressed. First, the stack's size may grow in an uncontrolled way. The solution to this problem is simple: once 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 $range(cW)$ (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 [2]. 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.

The second issue that needs to be addressed is how to efficiently compute $\texttt{repr}(\texttt{cW})$ from $\texttt{repr}(\texttt{W})$ for the characters c such that cW is right-maximal in T. In [2, 3] 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 the notation of [2], let us call $\texttt{rangeDistinct}(\texttt{i}, \texttt{j})$ the operation that returns all distinct characters in $BWT[i, j]$. 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 [2, 4] (even including implicit Weiner links[7]), globally the number of distinct characters returned by `rangeDistinct` operations is $O(n)$. An implementation of `rangeDistinct` on wavelet trees is discussed in [5] 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 11 of Algorithm 1). This function can be implemented by observing that

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

where $a = \texttt{chars}_\texttt{W}[\texttt{i}]$ for $1 \leq i < |\texttt{first}_\texttt{W}|$, and noting that $\texttt{left}(\texttt{Wa})$ and $\texttt{right}(\texttt{Wa})$ are available in $\texttt{repr}(\texttt{W})$. 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 $\texttt{repr}(\texttt{W})$. 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 $\texttt{repr}(\texttt{cW}) = \langle \texttt{first}_\texttt{cW}, |W| + 1 \rangle$ 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 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.

[7] 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).

B Notes on Beller et al.'s Algorithm

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 $LCP[R_i+1] = \perp$, and successively $LCP[R_i+1]$ is set to a value different than $\perp$. Clearly, this can happen at most n times. In [5] the authors moreover show that, even counting the left-extensions of those intervals (that we compute after popping each interval from the queue), the total number of computed 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).

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 in a (possibly empty) sequence with associated LCP value (i.e. the third element in the triples) $\ell+1$, followed by a sequence with associated LCP value ℓ, for some ℓ. We can therefore store the two sequences (with associated LCP value) independently, and there is no need to store the LCP values in the triples themselves (i.e. the queue's elements become just ranges). Note also that we pop elements from the sequence with associated LCP value ℓ, and push elements in the sequence with associated LCP value $\ell+1$. When the former sequence is empty, we create a new sequence with associated LCP value $\ell+2$ and start popping from the sequence with associated LCP value $\ell+1$ (and so on). Beller et al. represent each of the two sequences separately as follows. While inserting elements in a sequence, as long as the sequence's length 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. As soon as the sequence's length exceeds $n/\log n$, we switch to a representation that uses two packed bitvectors of length n storing, respectively, the left- and right-most boundaries of the ranges in the sequence. Note that this representation can be used because the sequence of intervals corresponds to suffix array ranges of strings of some fixed length ℓ, therefore there cannot be overlapping intervals. Pushing an interval in this new queue's representation takes constant time. Popping all the t intervals from one of the two sequences, on the other hand, can be implemented in $O(t + n/\log n)$ time by scanning the bitvectors (this requires using simple bitwise operations on words, see [5] for all details). Since at most $O(\log n)$ sequences will exceed size $n/\log n$, overall pop operations take amortized constant time.

C Proofs

▶ **Lemma 6.** *Algorithms 1 and 2 correctly compute 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 input and output.*

Proof.

Correctness and completeness - Algorithm 1. We start by proving that Beller et al.'s procedure in Line 2 of Algorithm 1 (procedure `BGOS(BWT,LCP)`) fills all the node-type LCP entries correctly. The proof proceeds by induction on the LCP value ℓ and follows the original proof of [5]. At the beginning, we insert in the queue all c-intervals, for $c \in \Sigma$. For each such interval $\langle L, R\rangle$ we set $LCP[R+1] = \ell = 0$. It is easy to see that after this step all and only the node-type LCP values equal to 0 are correctly set. Assume, by induction, that all node-type LCP values less than or equal to ℓ have been correctly set, and 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 $LCP[R'+1] \neq \perp$, then we have nothing to do. However, if $LCP[R'+1] = \perp$, then it must be the cases that (i)

the value to write in this cell satisfies $LCP[R'+1] \geq \ell+1$, since by induction we have already filled 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.

The above argument settles correctness; to prove completeness, assume that, at some point, $LCP[i] = \perp$ and the correct value to be written in this cell 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 set $LCP[i] \leftarrow \ell+1$. We proceed by induction on $|W|$. Note that we separately set 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 are set. From now on we can therefore assume that we are setting 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 completes the completeness proof since we showed that $\langle \texttt{range}(\texttt{bV}), \ell+1\rangle$ is in the queue, so sooner or later we will pop it, extend it with a, and set $\texttt{LCP}[\texttt{right}(\texttt{abV}) + \texttt{1}] = \texttt{LCP}[\texttt{i}] \leftarrow \ell+1$.

If the queue uses too much space, then Algorithm 1 switches to a stack and Lines 4-14 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 sets $LCP[\texttt{first}_\texttt{W}[\texttt{j}]] = LCP[i] \leftarrow \ell = |W|$.

Correctness and completeness - Algorithm 2. Proving correctness and completeness of this procedure is much easier. 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 set $LCP[L+1, R]$ to ℓ, the string depth of the corresponding string (excluding the final character #). It is easy to see that each leaf-type LCP value is correctly set 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 4, the number of Weiner links is linear. By the same analysis of Section 4, the operation in Line 11 takes $O(k \log \sigma)$ amortized time on wavelet trees, and sorting in Line 12 (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 9 we set $R - L - 2$ LCP values. As observed above, these intervals form a cover of $[1, n]$ and therefore Line 9 is executed no more than n times. Line 13 takes time $O(k \log \sigma)$. Finally, in Line 14 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 13-14 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. ◀

D Enumerating Suffix Tree Intervals in Succinct Space

We note that Algorithm 1 can be used to enumerate suffix tree intervals using just $o(n \log \sigma)$ space on top of the input BWT of a single text, when this is represented with a wavelet tree. This is true by definition in Belazzougui's procedure (Lines 4-14), but a closer look reveals that also Beller et al's procedure (Line 2) enumerates suffix tree intervals. At each step, we pop from the queue an element $\langle\langle L, R\rangle, |W|\rangle$ with $\langle L, R\rangle =$ `range(W)` for some string W, left-extend the range with all $a \in$ `BWT.rangeDistinct(L,R)`, obtaining the ranges `range(aW)` $= \langle L_a, R_a\rangle$ and, only if $LCP[R_a + 1] = \perp$, set $LCP[R_a + 1] \leftarrow |W|$ and push $\langle\langle L_a, R_a\rangle, |W| + 1\rangle$ on the stack. But then, since $LCP[R_a + 1] = |W|$ we have that the R_a-th and $(R_a + 1)$-th smallest suffixes start, respectively, with aUc and aUd for some $c < d \in \Sigma$, where $W = Uc$. This implies that aU is right-maximal, and the corresponding suffix tree node has at least two children labeled c and d; in particular, $\langle L_a, R_a\rangle$ is the range of $aW = aUc$, that is, one of these two children. Since we assume that we are working with a single text, $\langle L_a, R_a\rangle$ is the range of a suffix tree node if and only if $R_a > L_a$: in this case, we return this range. Completeness of the visit (i.e. we return all suffix tree nodes' intervals)

follows from the completeness of the LCP-array construction procedure (i.e. we fill all LCP values). To conclude note that, to perform our visit, we are using the array LCP to store null/non-null entries (i.e. $\perp$ or any other number); this shows that we do not actually need the LCP array: a bitvector of length $n = o(n \log \sigma)$ is sufficient (remember that Beller et al.'s strategy is used on large alphabets, so $n = o(n \log \sigma)$ holds).