# External memory BWT and LCP computation for sequence collections with applications

**Lavinia Egidi**[1]
University of Eastern Piedmont, Alessandria, Italy
lavinia.egidi@uniupo.it
https://orcid.org/0000-0002-9745-0942

**Felipe A. Louza**[2]
Department of Computing and Mathematics, University of São Paulo, Ribeirão Preto, Brazil
louza@usp.br
https://orcid.org/0000-0003-2931-1470

**Giovanni Manzini**[3]
University of Eastern Piedmont, Alessandria, Italy
IIT CNR, Pisa, Italy
giovanni.manzini@uniupo.it
https://orcid.org/0000-0002-5047-0196

**Guilherme P. Telles**[4]
Institute of Computing, University of Campinas, Campinas, Brazil
gpt@ic.unicamp.br

## Abstract

We propose an external memory algorithm for the computation of the BWT and LCP array for a collection of sequences. Our algorithm takes the amount of available memory as an input parameter, and tries to make the best use of it by splitting the input collection into subcollections sufficiently small that it can compute their BWT in RAM using an optimal linear time algorithm. Next, it merges the partial BWTs in external memory and in the process it also computes the LCP values. We show that our algorithm performs $\mathcal{O}(n\,\mathsf{maxlcp})$ sequential I/Os, where $n$ is the total length of the collection and $\mathsf{maxlcp}$ is the maximum LCP value. The experimental results show that our algorithm outperforms the current best algorithm for collections of sequences with different lengths and when the *average* LCP of the collection is relatively small compared to the length of the sequences.

In the second part of the paper, we show that our algorithm can be modified to output two additional arrays that, combined with the BWT and LCP arrays, provide simple, scan based, external memory algorithms for three well known problems in bioinformatics: the computation of the all pairs suffix-prefix overlaps, the computation of maximal repeats, and the construction of succinct de Bruijn graphs.

**2012 ACM Subject Classification** Theory of computation → Design and analysis of algorithms

**Keywords and phrases** Burrows-Wheeler Transform, Longest Common Prefix Array, All pairs suffix-prefix overlaps, Succinct de Bruijn graph, Maximal repeats

**Digital Object Identifier** 10.4230/LIPIcs.WABI.2018.10

[1] L.E. was partially supported by the University of Eastern Piedmont project *Behavioural Types for Dependability Analysis with Bayesian Networks*
[2] F.A.L. was supported by the grant #2017/09105-0 from the São Paulo Research Foundation (FAPESP)
[3] G.M. was partially supported by PRIN grant 201534HNXC
[4] G.P.T. acknowledges the support of CNPq

**Supplement Material** Source code: `https://github.com/felipelouza/egap/`

**Acknowledgements** We wish to thank the anonymous referees for useful observations; we appreciated in particular an extensive comment on external memory models.

## 1 Introduction

A fundamental problem in bioinformatics is the ability to efficiently search into the billions of DNA sequences produced by NGS studies. The Burrows Wheeler transform (BWT) is a well known structure which is the starting point for the construction of compressed indices for collection of sequences [24]. The BWT is often complemented with the Longest Common Prefix (LCP) array since the latter makes it possible to efficiently emulate Suffix Tree algorithms [14, 31]. The construction of such data structures is a challenging problem. Although the final outcome is a *compressed* index, construction algorithms can be memory hungry and the necessity of developing *lightweight*, i.e. space economical, algorithms was recognized since the very beginning of the field [9, 26, 27]. When even lightweight algorithms do not fit in RAM, one has to resort to external memory construction algorithms (see [13, 18, 19, 23] and references therein).

Although the space efficient computation of the BWT in RAM is well studied, and remarkable advances have been recently obtained [2, 30], for external memory computation the situation is less satisfactory. For collections of sequences, the first external memory algorithm is the BCR algorithm described in [1] that computes the multi-string BWT for a collection of total size $n$, performing a number of sequential I/Os proportional to $nK$, where $K$ is the length of the longest sequence in the collection. This approach is clearly not competitive when the sequences have non homogeneous lengths, and it is far from the theoretical optimal even for sequences of equal length. Nevertheless, the simplicity of the algorithm makes it very effective for collections of relatively short sequences, and this has become the reference tool for this problem. This approach was later extended [11] to compute also the LCP values with the same asymptotic number of I/Os. When computing also the LCP values, or when the input strings have different lengths, the algorithm uses $\mathcal{O}(m)$ words of RAM, where $m$ is the number of input sequences.

In this paper, we present a new external memory algorithm for the computation of the BWT and LCP array for a collection of sequences. Our algorithm takes the amount of available RAM as an input parameter, and tries to make the best use of it by splitting the input into subcollections sufficiently small so that it can compute their BWT in internal memory using an optimal linear time algorithm. Next, it merges the partial BWTs in external memory and in the process it also computes the LCP values. Since the LCP values are computed in a non-standard order, the algorithm is completed by an external memory merge sort procedure that computes the final LCP array. We show that our algorithm performs a number of sequential I/Os between $\mathcal{O}(n\,\mathsf{avelcp})$ and $\mathcal{O}(n\,\mathsf{maxlcp})$, where $\mathsf{avelcp}$ and $\mathsf{maxlcp}$ are respectively the average and the maximum Longest Common Prefix of the input sequences. The experimental results show that our algorithm is indeed much faster than BCR for collections of sequences when the average LCP is relatively small compared to the length of the sequences.

To our knowledge, the only other known external memory algorithm for computing the BWT and LCP arrays of a collection of sequences is the one recently proposed in [4] that performs $\mathcal{O}(n\,\mathsf{maxlcp})$ sequential I/Os and uses $\mathcal{O}(m + k)$ words of RAM. We plan to experimentally compare this algorithm to ours in the near future.

Another contribution of the paper, which follows from our main result, is the design of simple external memory algorithms for three well known problems, namely: the computation of maximal repeats [21, 34], the computation of the all pairs suffix-prefix overlaps [16, 33, 35], and the construction of succinct de Bruijn graphs [3, 7, 8]. This is achieved using the BWT and LCP arrays, together with two additional arrays that our algorithm can compute without any asymptotic slowdown. The first one is the so called Document Array providing for each suffix the ID of the sequence it belongs to; the second one is a bit array indicating whether each suffix is a substring of the one immediately following it in lexicographic order. Our external memory algorithms for these problems are derived from known internal memory algorithms, but they process the input data in a single sequential scan. In addition, for the problem of the all pairs suffix-prefix, we go beyond the recent solutions [5, 33, 35] that compute *all* the overlaps, by computing only the overlaps above a certain length, still spending constant time per reported overlap. Since the above problems often involve huge datasets we believe it is important to provide external memory algorithms. To our knowledge, only for the all pair suffix-prefix problem there exists an external memory algorithm that computes all the overlaps given the BWT, LCP and Generalized Suffix Array of the input collection [5, Algorithm 2].

## 2 Background

Let $\mathsf{s}[1,n]$ denote a string of length $n$ over an alphabet $\Sigma$ of size $\sigma$. As usual, we assume $\mathsf{s}[n]$ is a special symbol (end-marker) not appearing elsewhere in $\mathsf{s}$ and lexicographically smaller than any other symbol. We write $\mathsf{s}[i,j]$ to denote the substring $\mathsf{s}[i]\mathsf{s}[i+1]\cdots\mathsf{s}[j]$. If $j \geq n$ we assume $\mathsf{s}[i,j] = \mathsf{s}[i,n]$. If $i > j$ or $i > n$ then $\mathsf{s}[i,j]$ is the empty string. Given two strings $\mathsf{s}_1$ and $\mathsf{s}_2$ we write $\mathsf{s}_1 \preceq \mathsf{s}_2$ ($\mathsf{s}_1 \prec \mathsf{s}_2$) to denote that $\mathsf{s}_1$ is lexicographically (strictly) smaller than $\mathsf{s}_2$. We denote by $\mathsf{LCP}(\mathsf{s}_1, \mathsf{s}_2)$ the length of the longest common prefix between $\mathsf{s}_1$ and $\mathsf{s}_2$.

The *suffix array* $\mathsf{sa}[1,n]$ associated to $\mathsf{s}$ is the permutation of $[1,n]$ giving the lexicographic order of $\mathsf{s}$'s suffixes, that is, for $i = 1,\dots,n-1$, $\mathsf{s}[\mathsf{sa}[i],n] \prec \mathsf{s}[\mathsf{sa}[i+1],n]$.

The *longest common prefix* array $\mathsf{lcp}[1,n+1]$ is defined for $i = 2,\dots,n$ by

$$\mathsf{lcp}[i] = \mathsf{LCP}(\mathsf{s}[\mathsf{sa}[i-1],n], \mathsf{s}[\mathsf{sa}[i],n]); \tag{1}$$

the $\mathsf{lcp}$ array stores the length of the longest common prefix (LCP) between lexicographically consecutive suffixes. For convenience we define $\mathsf{lcp}[1] = \mathsf{lcp}[n+1] = -1$.

Let $\mathsf{s}_1[1,n_1],\dots,\mathsf{s}_k[1,n_k]$ be such that $\mathsf{s}_1[n_1] = \$_1,\dots,\mathsf{s}_k[n_k] = \$_k$, where where $\$_1 < \dots < \$_k$ are $k$ symbols not appearing elsewhere in $\mathsf{s}_1,\dots,\mathsf{s}_k$ and smaller than any other symbol. Let $\mathsf{sa}_{1\cdots k}[1,n]$ denote the suffix array of the concatenation $\mathsf{s}_1\cdots\mathsf{s}_k$ of total length $n = \Sigma_{h=1}^{k} n_h$. The *multi-string* BWT [11, 25] of $\mathsf{s}_1,\dots,\mathsf{s}_k$, denoted by $\mathsf{bwt}_{1\cdots k}[1,n]$, is defined as

$$\mathsf{bwt}_{1\cdots k}[i] = \begin{cases} \mathsf{s}_j[\mathsf{n}_j] & \text{if } \mathsf{sa}_{1\cdots k}[i] = \Sigma_{h=1}^{j-1} n_h + 1 \\ \mathsf{s}_j[\mathsf{sa}_{1\cdots k}[i] - \Sigma_{h=1}^{j-1} n_h - 1] & \text{if } \Sigma_{h=1}^{j-1} n_h + 1 < \mathsf{sa}_{1\cdots k}[i] \leq \Sigma_{h=1}^{j} n_h. \end{cases} \tag{2}$$

Essentially $\mathsf{bwt}_{1\cdots k}$ is a permutation of the symbols in $\mathsf{s}_1,\dots,\mathsf{s}_k$ such that the position in $\mathsf{bwt}_{1\cdots k}$ of $\mathsf{s}_i[j]$ is given by the lexicographic rank of its context $\mathsf{s}_i[j+1,n_i]$ (or $\mathsf{s}_i[1,n_i]$ if $j = n_i$). Fig. 1 shows an example with $k = 2$. Notice that for $k = 1$, this is the usual Burrows-Wheeler transform [10].

Given the suffix array $\mathsf{sa}_{1\cdots k}[1,n]$ of the concatenation $\mathsf{s}_1\cdots\mathsf{s}_k$, we consider the corresponding LCP array $\mathsf{lcp}_{1\cdots k}[1,n]$ defined as in (1) (see again Fig. 1). Note that, for $i = 2,\dots,n$, $\mathsf{lcp}_{1\cdots k}[i]$ gives the length of the longest common prefix between the contexts of $\mathsf{bwt}_{1\cdots k}[i]$

| lcp | bwt | *context* |
|---|---|---|
| -1 | b | $\$_1$ |
| 0 | c | ab$\$_1$ |
| 2 | $\$_1$ | abcab$\$_1$ |
| 0 | a | b$\$_1$ |
| 1 | a | bcab$\$_1$ |
| 0 | b | cab$\$_1$ |
| -1 | | |

| lcp | bwt | *context* |
|---|---|---|
| -1 | c | $\$_2$ |
| 0 | $\$_2$ | aabcabc$\$_2$ |
| 1 | c | abc$\$_2$ |
| 3 | a | abcabc$\$_2$ |
| 0 | a | bc$\$_2$ |
| 2 | a | bcabc$\$_2$ |
| 0 | b | c$\$_2$ |
| 1 | b | cabc$\$_2$ |
| -1 | | |

| id | $\mathsf{lcp}_{12}$ | $\mathsf{bwt}_{12}$ | *context* |
|---|---|---|---|
| 1 | -1 | b | $\$_1$ |
| 2 | 0 | c | $\$_2$ |
| 2 | 0 | $\$_2$ | aabcabc$\$_2$ |
| 1 | 1 | c | ab$\$_1$ |
| 2 | 2 | c | abc$\$_2$ |
| 1 | 3 | $\$_1$ | abcab$\$_1$ |
| 2 | 5 | a | abcabc$\$_2$ |
| 1 | 0 | a | b$\$_1$ |
| 2 | 1 | a | bc$\$_2$ |
| 1 | 2 | a | bcab$\$_1$ |
| 2 | 4 | a | bcabc$\$_2$ |
| 2 | 0 | b | c$\$_2$ |
| 1 | 1 | b | cab$\$_1$ |
| 2 | 3 | b | cabc$\$_2$ |
| | -1 | | |

**Figure 1** LCP array and BWT for $\mathsf{s}_1 = \mathsf{abcab}\$_1$ and $\mathsf{s}_2 = \mathsf{aabcabc}\$_2$, and multi-string BWT and corresponding LCP array for the same strings. Column id shows, for each entry of $\mathsf{bwt}_{12} = \mathsf{bc}\$_2\mathsf{cc}\$_1\mathsf{aaaabbb}$ whether it comes from $\mathsf{s}_1$ or $\mathsf{s}_2$.

and $\mathsf{bwt}_{1\cdots k}[i-1]$. We stress that all practical implementations use a single \$ symbol as end-marker for all strings to avoid alphabet explosion, but end-markers from different strings are then sorted as described, i.e., on the basis of the index of the strings they belong to.

## 2.1 Computing multi-string BWTs

The gSACA-K algorithm [22], based on algorithm SACA-K [32], computes the suffix array for a string collection. Given a collection of strings of total length $n$, gSACA-K computes the suffix array for their concatenation in $O(n)$ time using $(\sigma + 1)\log n$ additional bits (in practice, only 2KB are used for ASCII alphabets). It is optimal for alphabets of constant size $\sigma = O(1)$. The *multi-string* $\mathsf{bwt}_{1\cdots k}$ of $\mathsf{s}_1, \ldots, \mathsf{s}_k$ can be easily obtained from the suffix array as in (2). gSACA-K can compute also the lcp array $\mathsf{lcp}_{1\cdots k}$ still in linear time using only the additional space for the lcp values.

## 2.2 Merging multi-string BWTs

The Gap algorithm [12], based on an earlier algorithm by Holt and McMillan [17], is a simple procedure to merge multi-string BWTs. In its original formulation the Gap algorithm can also merge LCP arrays, but in this paper we compute LCP values using a different approach more suitable for external memory execution. We describe here only the main idea behind Gap and refer the reader to [12] for further details.

Given $k$ multi-string BWTs for disjoint subcollections, the Gap algorithm computes a multi-string BWT for the whole collection. The computation does not explicitly need the collection but only the multi-string BWTs to be merged. For simplicity in the following we assume we are merging $k$ single-string BWTs $\mathsf{bwt}_1 = \mathsf{bwt}(\mathsf{s}_1), \ldots, \mathsf{bwt}_k = \mathsf{bwt}(\mathsf{s}_k)$; the algorithm does not change in the general case where the inputs are multi-string BWTs. Recall that computing $\mathsf{bwt}_{1\cdots k}$ amounts to sorting the symbols of $\mathsf{bwt}_1, \ldots, \mathsf{bwt}_k$ according to the lexicographic order of their contexts, where the context of symbol $\mathsf{bwt}_j[i]$ is $\mathsf{s}_j[\mathsf{sa}_j[i], n_j]$, for $j = 1, \ldots, k$.

The Gap algorithm works in successive iterations. After the $h$-th iteration the entries of each $\mathsf{bwt}_\lambda$ are sorted on the basis of the first $h$ symbols of their context. More formally, the output of the $h$-th iteration is a $k$-valued vector $Z^{(h)}$ containing $n_\lambda = |\mathsf{s}_\lambda|$ entries $\lambda$ for each $\lambda = 1, \ldots, k$, such that the following property holds.

▶ **Property 1.** *For $\lambda_1, \lambda_2 \in \{1, \ldots, k\}$, $\lambda_1 < \lambda_2$, and $i = 1, \ldots, n_{\lambda_1}$ and $j = 1, \ldots, n_{\lambda_2}$ the $i$-th $\lambda_1$ precedes the $j$-th $\lambda_2$ in $Z^{(h)}$ iff $\mathsf{s}_{\lambda_1}[\mathsf{sa}_{\lambda_1}[i], \mathsf{sa}_{\lambda_1}[i]+h-1] \preceq \mathsf{s}_{\lambda_2}[\mathsf{sa}_{\lambda_2}[j], \mathsf{sa}_{\lambda_2}[j]+h-1]$.* ◀

Following Property 1 we identify the $i$-th $\lambda$ in $Z^{(h)}$ with $\mathsf{bwt}_\lambda[i]$ so that $Z^{(h)}$ corresponds to a permutation of $\mathsf{bwt}_{1\cdots k}$. Property 1 is equivalent to state that we can logically partition $Z^{(h)}$ into $b(h) + 1$ blocks

$$Z^{(h)}[1, \ell_1],\ Z^{(h)}[\ell_1 + 1, \ell_2],\ \ldots,\ Z^{(h)}[\ell_{b(h)} + 1, n] \tag{3}$$

such that each block is either a singleton or corresponds to the set of $\mathsf{bwt}_{1\cdots k}$ symbols whose contexts are prefixed by the same length-$h$ string. Within each block, for $\lambda_1 < \lambda_2$, the symbols of $\mathsf{bwt}_{\lambda_1}$ precede those of $\mathsf{bwt}_{\lambda_2}$ and the context of any symbol in block $Z^{(h)}[\ell_j + 1, \ell_{j+1}]$ is lexicographically smaller than the context of any symbol in block $Z^{(h)}[\ell_k + 1, \ell_{k+1}]$ with $k > j$. We keep explicit track of such blocks using a bit array $B[1, n + 1]$ such that at the end of iteration $h$ it is $B[i] \neq 0$ if and only if a block of $Z^{(h)}$ starts at position $i$, i.e. $\mathsf{lcp}_{1\cdots k}[i] \leq h - 1$. By Property 1, when all entries in $B$ are nonzero, $Z^{(h)}$ describes how the $\mathsf{bwt}_j$ $(j = 1, \ldots, k)$ should be merged to get $\mathsf{bwt}_{1\cdots k}$.

## 3 The eGap algorithm

At a glance, the eGap algorithm for computing the multi-string BWT and LCP array in external memory works in three phases. First it builds multi-string BWTs for sub-collections in internal memory, then it merges these BWTs in external memory and generates the LCP values. Finally, it merges the LCP values in external memory.

### 3.1 Phase 1: BWT computation

Given a collection of sequences $\mathsf{s}_1, \mathsf{s}_2, \ldots, \mathsf{s}_k$, we split it into sub-collections sufficiently small that we can compute the multi-string SA for each one of them using the linear time internal memory gSACA-K algorithm (Section 2). After computing each SA, Phase 1 writes each multi-string BWT to disk in uncompressed form using one byte per character.

### 3.2 Phase 2: BWT merging and LCP computation

This part of the algorithm is based on the Gap algorithm described in Section 2 but it is designed to work efficiently in external memory and it computes LCP values in addition to merging the input (multi-string) BWTs. In the following we assume that the input consists of $k$ BWTs $\mathsf{bwt}_1, \ldots, \mathsf{bwt}_k$ of total length $n$ over an alphabet of size $\sigma$. The input BWTs are read from disk and never moved to internal memory. We denote by $\mathsf{bwt}_{1\cdots k}$ and $\mathsf{lcp}_{1\cdots k}$ the output BWT and LCP arrays.

The algorithm initially sets $Z^{(0)} = \mathbf{1}^{n_1}\mathbf{2}^{n_2}\ldots\mathbf{k}^{n_k}$ and $B = \mathbf{1}\mathbf{0}^{n-1}\mathbf{1}$. Since the context of every symbol is prefixed by the same length-0 string (the empty string), initially there is a single block containing all symbols. At iteration $h$ the algorithm computes $Z^{(h)}$ from $Z^{(h-1)}$ as follows. We define an array $F[1, \sigma]$ such that $F[c]$ contains the number of occurrences of characters smaller than $c$ in $\mathsf{bwt}_{1\cdots k}$. $F$ partitions $Z^{(h)}$ into $\sigma$ buckets, one for each symbol. Using $Z^{(h-1)}$ we scan the partially merged BWT, and whenever we encounter the BWT character $c$ coming from $\mathsf{bwt}_i$, with $i \in \{1, \ldots, k\}$, we store it in the next free position of bucket $c$ in $Z^{(h)}$; note that $c$ is not actually moved, instead we write $i$ in its corresponding position in $Z^{(h)}$. Instead of using distinct arrays $Z^{(0)}, Z^{(1)}, \ldots$ we only use two arrays $Z^{\mathsf{old}}$ and $Z^{\mathsf{new}}$, that are kept on disk. At the beginning of iteration $h$ it is $Z^{\mathsf{old}} = Z^{(h-1)}$ and $Z^{\mathsf{new}} = Z^{(h-2)}$; at the end $Z^{\mathsf{new}} = Z^{(h)}$ and the roles of the two files are swapped. While $Z^{\mathsf{old}}$ is accessed sequentially, $Z^{\mathsf{new}}$ is updated sequentially within each bucket, that is within each set of positions corresponding to a given character. Since the boundary of each bucket is known in advance we logically split the $Z^{\mathsf{new}}$ file in buckets and write to each one sequentially.

The key to the computation of the LCP array by eGap is to exploit the bitvector $B$ used by Gap to mark the beginning of blocks. We observe that entry $B[i]$ is set to $\mathbf{1}$ during iteration $h = \mathsf{lcp}_{1\cdots k}[i] + 1$, when it is determined that the contexts of $\mathsf{bwt}_{1\cdots k}[i]$ and $\mathsf{bwt}_{1\cdots k}[i-1]$ have a common prefix of length exactly $h-1$ (and a new block is created). We introduce an additional bit array $B_x$ such that, at the beginning of iteration $h$, $B_x[i] = \mathbf{1}$ iff $B[i]$ has been set to $\mathbf{1}$ at iteration $h-2$ or earlier. During iteration $h$, if $B[i] = \mathbf{1}$ we look at $B_x[i]$. If $B_x[i] = \mathbf{0}$ then $B[i]$ has been set at iteration $h-1$: thus we output to a temporary file $F_{h-2}$ the pair $\langle i, h-2\rangle$ to record that $\mathsf{lcp}_{1\cdots k}[i] = h-2$, then we set $B_x[i] = \mathbf{1}$ so no pair for position $i$ will be produced in the following iterations. At the end of iteration $h$, file $F_{h-2}$ contains all pairs $\langle i, \mathsf{lcp}_{1\cdots k}[i]\rangle$ with $\mathsf{lcp}[i] = h-2$; the pairs are written in increasing order of their first component, since $B$ and $B_x$ are scanned sequentially. These temporary files will be merged in Phase 3.

As proven in [12, Lemma 7], if at iteration $h$ of the Gap algorithm we set $B[i] = \mathbf{1}$, then at any iteration $g \geq h+2$ processing the entry $Z^{(g)}[i]$ will not change the arrays $Z^{(g+1)}$ and $B$. Since the roles of the $Z^{\mathsf{old}}$ and $Z^{\mathsf{new}}$ files are swapped at each iteration, and at iteration $h$ we scan $Z^{\mathsf{old}} = Z^{(h-1)}$ to update $Z^{\mathsf{new}}$ from $Z^{(h-2)}$ to $Z^{(h)}$, we can compute only the entries $Z^{(h)}[j]$ that are different from $Z^{(h-2)}[j]$. In particular, any range $[\ell, m]$ such that $B_x[\ell] = B_x[\ell+1] = \cdots = B_x[m] = \mathbf{1}$ can be added to a set of *irrelevant* ranges that the algorithm may skip in successive iterations (irrelevant ranges are defined in terms of the array $B_x$ as opposed to the array $B$, since before skipping an irrelevant range we need to update both $Z^{\mathsf{old}}$ and $Z^{\mathsf{new}}$). We read from one file the ranges to be skipped at the current iteration and simultaneously write to another file the ranges to be skipped at the next iteration (note that irrelevant ranges are created and consumed sequentially). Since skipping a single irrelevant range takes $\mathcal{O}(k+\sigma)$ time, an irrelevant range is stored only if its size is larger than a given threshold $t$ and we merge consecutive irrelevant ranges whenever possible. In our experiments we used $t = \max(256, k+\sigma)$. In the worst case the space for storing irrelevant ranges could be $\mathcal{O}(n)$ but in actual experiments it was always less than $0.1n$ bytes.

As in the Gap algorithm, when all entries in $B$ are nonzero, $Z^{\mathsf{old}}$ describes how the BWTs $\mathsf{bwt}_j$ $(j = 1, \ldots, k)$ should be merged to get $\mathsf{bwt}_{1\cdots k}$, and a final sequential scan of the input BWTs along with $Z^{\mathsf{old}}$ allows to write $\mathsf{bwt}_{1\cdots k}$ to disk, in sequential order. Our implementation can merge at most $2^7 = 128$ BWTs at a time because we use 7 bits to store each entry of $Z^{\mathsf{old}}$ and $Z^{\mathsf{new}}$. These arrays are maintained on disk in two separate files; the additional bit of each byte are used to keep the current and the next copy of $B$. The bit array $B_x$ is stored separately in a file of size $n/8$ bytes. To merge of a set of $k > 128$ BWTs we split the input in subsets of cardinality 128 and merge them in successive rounds. We have also implemented a semi-external version of the merge algorithm that uses $n$ bytes of RAM. The $i$-th byte is used to store $Z^{\mathsf{old}}[i]$ and $Z^{\mathsf{new}}[i]$ (3 bits each), $B[i]$ and $B_x[i]$.

### 3.3 Phase 3: LCP merging

At the end of Phase 2 all LCP-values have been written to the temporary files $F_h$ on disk as pairs $\langle i, \mathsf{lcp}[i]\rangle$. Each file $F_h$ contains all pairs with second component equal to $h$ in order of increasing first component. The computation of the LCP array is completed using a standard external memory multiway merge [20, ch. 5.4.1] of $\mathsf{maxlcp}$ sorted files, where $\mathsf{maxlcp} = \max_i(\mathsf{lcp}_{1\cdots k}[i])$ is the largest LCP value.

### 3.4 Analysis

During Phase 1, gSACA-K computes the suffix array for a sub-collection of total length $m$ using $9m$ bytes. If the available RAM is $M$, the input is split into subcollections of size $\approx M/9$. Since gSACA-K runs in linear time, if the input collection has total size $n$, Phase 1 takes $\mathcal{O}(n)$ time overall.

A single iteration of Phase 2 consists of a complete scan of $Z^{(h-1)}$ except for the irrelevant ranges. Since the algorithm requires $\mathsf{maxlcp}$ iterations, without skipping the irrelevant ranges the algorithm would require $\mathsf{maxlcp}$ sequential scans of $\mathcal{O}(n)$ items. Reasoning as in [12, Theorem 8] we get that by skipping irrelevant ranges the overall amount of data *directly* read/written by the algorithm is $\mathcal{O}(n\,\mathsf{avelcp})$ items where $\mathsf{avelcp}$ is the arithmetic average of the entries in the final LCP array. However, if we reason in terms of disk blocks, every time we skip an irrelevant range we discard the current block and load a new one (unless the beginning of the new relevant range is inside the same block, in that case skipping the irrelevant range does not save any I/O). We can upper bound this extra cost, with an overhead of $\mathcal{O}(1)$ blocks for each irrelevant range skipped. Summing up, assuming the total number of skipped ranges is $Ir$ and that each disk block consists of $S$ words, the I/O complexity of Phase 2 is $\mathcal{O}(Ir + n\,\mathsf{avelcp}\log k/(S\log n))$ block I/Os, where $k$ is the number of input BWTs. Although the preliminary experiments in Section 4 suggest that in practice $Ir$ is small, for simplicity and uniformity with the previous literature we upper bound the cost of Phase 2 with $\mathcal{O}(n\,\mathsf{maxlcp})$ sequential I/Os, (corresponding to $\mathcal{O}(n\,\mathsf{maxlcp}\log k/(S\log n))$ block I/Os). As a future work, we plan to do a detailed theoretical and experimental analysis of the impact of skipping irrelevant ranges.

Phase 3 takes $\mathcal{O}(\lceil\log_K \mathsf{maxlcp}\rceil)$ rounds; each round merges $K$ LCP files by sequentially reading and writing $\mathcal{O}(n)$ bytes of data. The overall cost of Phase 3 is therefore $\mathcal{O}(n\log_K \mathsf{maxlcp})$ sequential I/Os. In our experiments we used $K = 256$; since in our tests $\mathsf{maxlcp} < 2^{16}$ two merging rounds were always sufficient.

The above analysis suggests that Phase 2 is the most expensive phase of the eGap algorithm. Indeed, in our experiments we found that Phase 2 always took at least 95% of the overall running time.

## 4 Experiments

In this section we report some preliminary experiments on the eGap algorithm. Testing external memory algorithms is extremely time consuming since, to make a realistic external memory setting, one has to use an amount of RAM smaller than the size of the data. If more RAM is available, even if the algorithm is supposedly not using it, the operating system will use it to temporary store disk data and the algorithm will be no longer really working in external memory. This phenomenon will be apparent also from our experiments. For this reasons we used datasets of size 8GB, reported in Table 1, and a machine with 32GB of RAM but reduced at boot time to 1GB, to simulate input data much larger than the available

**Table 1** Datasets used in our experiments. shortreads are DNA reads from human genome[5] trimmed to length 100. longreads are Illumina HiSeq 4000 paired-end RNA-seq reads from plant Setaria viridis[6] trimmed to length 300. pacbio are PacBio RS II reads from Triticum aestivum (wheat) genome[7] with different lengths. pacbio.1000 are the strings from pacbio trimmed to length 1,000. Columns 5 and 6 show the maximum and average lengths of the single strings. Columns 7 and 8 show the maximum and average LCPs of the collections.

| Name | Size GB | $\sigma$ | N. of strings | Max Len | Ave Len | Max LCP | Ave LCP |
|---|---|---|---|---|---|---|---|
| shortreads | 8.0 | 6 | 85,899,345 | 100 | 100 | 99 | 27.90 |
| longreads | 8.0 | 5 | 28,633,115 | 300 | 300 | 299 | 90.28 |
| pacbio.1000 | 8.0 | 5 | 8,589,934 | 1,000 | 1,000 | 876 | 18.05 |
| pacbio | 8.0 | 5 | 942,248 | 71,561 | 9,116 | 3,084 | 18.32 |

RAM, and 8GB, to simulate input data of approximately the same size as the available RAM and test also the semi external version of our algorithm. Note that for a 8GB input, the output BWT+LCP data has size 24GB, so even 8GB RAM is still significantly less than the input and the output combined.

We implemented eGap in ANSI C based on the source code of Gap [12] and gSACA-K [22]. Our algorithm was compiled with GNU GCC ver. 4.6.3, with optimizing option -O3. The source code is freely available at `https://github.com/felipelouza/egap/`. The experiments were conducted on a machine with GNU/Linux Debian 7.0/64 bits operating system using an Intel i7-3770 3.4 GHz processor with 8 MB cache, 32 GB of RAM and a 2.0 TB SATA hard disk with 7200 RPM and 64 MB cache.

We compared eGap with BCR+LCP[8] from the BEETL Library [1] which is the currently most used tool for the construction of BWT and LCP arrays in external memory. As a reference we also tested the external memory tool eGSA [23] that computes the Suffix and LCP arrays for a collection of sequences. However, we tested eGSA only using 32GB of RAM since the authors in [23] showed its running time degrades about 25 times when the RAM is restricted to the input size.

The results in Table 2 show that eGap's running time per input byte is roughly proportional to the average LCP. For example, if we look at the two pacbio datasets we see that they have widely different maximum LCPs, yet their running times are very close similarly to their average LCPs. According to the theoretical analysis in [1], BCR+LCP running time per input byte is proportional to the sequence length. Using 1GB RAM the tool BCR+LCP could not handle the shortreads dataset because of insufficient internal memory, and it stopped with an internal error after four days of computation on the pacbio.1000 dataset (we have contacted the authors about the error and plan to complete the comparison as soon as we obtain a stable version of the software). The tool BCR+LCP cannot handle input sequences of different lengths, so for the pacbio dataset we used the tool extLCP [11] by the same authors. However, extLCP appears to be not competitive on pacbio and for each instance we stopped it when it became clear that its running time was much higher than eGap's. In the future we also plan to include in the comparison two recent algorithms proposed in [4, 6]. In particular, the algorithm in [4] is also based on a merge strategy. First, for each $h$, it computes the sequence

5 `ftp://ftp.sra.ebi.ac.uk/vol1/ERA015/ERA015743/srf/`
6 `https://trace.ncbi.nlm.nih.gov/Traces/sra/?run=ERR1942989`
7 `https://trace.ncbi.nlm.nih.gov/Traces/sra/?run=SRR5816161`
8 `https://github.com/BEETL/BEETL/`

**Table 2** Running times in $\mu$ seconds per input byte.

| Name | eGap | | | BCR+LCP | | | eGSA |
|---|---|---|---|---|---|---|---|
| | 1GB | 8GB | 32GB | 1GB | 8GB | 32GB | 32GB |
| shortreads | 17.19 | 3.76 | 2.87 | $\times$ | 5.65 | 3.96 | 2.08 |
| longreads | 52.39 | 9.75 | 6.76 | 18.54 | 16.01 | 10.88 | 1.89 |
| pacbio.1000 | 24.88 | 3.54 | 1.81 | $\times$ | 54.00 | 36.96 | 1.89 |
| pacbio | 23.43 | 3.42 | 1.82 | $> 70$ | $> 50$ | $> 50$ | 1.74 |

of characters preceding all length-$h$ suffixes, ordered according to the lexicographical order of the length-$h$ suffixes (a sort of partial BWT). Then, it uses a procedure inspired by the Holt-Macmillan algorithm [17] to merge all the partial BWTs to the final output.

Although incomplete, the results show that BCR+LCP is competitive for short reads or collections with a large average LCP, while eGap clearly dominates in datasets with long reads and relatively small average LCP. In particular, when the available RAM is at least equal to the size of the input, eGap can use the semi-external strategy described in Section 3 and becomes significantly faster. Note that using 32GB RAM both algorithms become much faster: even though they allocate for their use a small fraction of that RAM, the operating system uses the remaining RAM as a buffer and avoids many disk accesses. Using 32GB RAM eGSA turns out to be the fastest algorithm and its running time appears to be less influenced by the size of the average LCP. Another advantage is that it also computes the Suffix Array, but it has the drawback of using a large amount of disk working space: 340GB for a 8GB input vs 56GB used by eGap.

## 5 Applications

In this section we show that the eGap algorithm, in addition to the BWT and LCP arrays, can output additional information that can be used to design efficient *external memory* algorithms for three well known problems on sequence collections: the computation of maximal repeats, the all pairs suffix-prefix overlaps, and the construction of succinct de Bruijn graphs. For these problems we describe algorithms which are derived from known (internal memory) algorithms but they process the input data in a single sequential scan. In addition, the amount of RAM used by the algorithms is usually much smaller than the size of inputs since it grows linearly with the number of sequences and the maximum LCP value.

Our starting point is the observation that the eGap algorithm can also output an array which provides, for each bwt entry, the id of the sequence to which that entry belongs. In information retrieval this is usually called the Document Array, so in the following we will denote it by da. In Phase 1 the gSACA-K algorithm can compute the da together with the lcp and bwt using only additional $4n$ bytes of space to store the da entries. These partial da's can be merged in Phase 2 using the $Z^{\mathsf{new}}$ array in the same way as the BWT entries. In the following we use bwt, lcp, and da to denote the multistring BWT, LCP and Document Array of a collection of $m$ sequences of total length $n$. We write s to denote the concatenation $\mathsf{s}_1 \cdots \mathsf{s}_m$ and sa to denote the suffix array of s. We will use s and sa to prove the correctness of our algorithms, but neither s nor sa are used in the computations.

### 5.1 Computation of Maximal Repeats

Different notions of maximal repeats have been used in the bioinformatic literature to model different notions of repetitive structure. We use a notion of maximal repeat from [15, Ch. 7]: we say that a string $\alpha$ is a *Type 1 maximal repeat* if $\alpha$ occurs in the collection at least twice and every extension, i.e. $c\alpha$ or $\alpha c$ with $c \in \Sigma$, occurs fewer times. We consider also a more restrictive notion: we say that a string $\alpha$ is a *Type 2 maximal repeat* if $\alpha$ occurs in the collection at least twice and every extension of $\alpha$ occurs at most once.

We first show how to compute Type 1 maximal repeats with a sequential scan of the arrays $\mathsf{bwt}$ and $\mathsf{lcp}$. The crucial observation is that we have a substring of length $\ell$ that prefixes $\mathsf{sa}$ entries $j, j+1, \ldots, i$ iff $\mathsf{lcp}[h] \geq \ell$ for $h = j+1, \ldots, i$, and both $\mathsf{lcp}[j]$ and $\mathsf{lcp}[i+1]$ are smaller than $\ell$. To ensure that the repeat is maximal, we also require that there exists $h \in [j+1, i]$ such that $\mathsf{lcp}[h] = \ell$ and that $\mathsf{bwt}[j, i]$ contains at least two distinct characters.

During the scan, we maintain a stack containing pairs $\langle j, \mathsf{lcp}[h]\rangle$ such that $j \leq h$; in addition if$\langle j', \mathsf{lcp}[h']\rangle$ is below $\langle j, \mathsf{lcp}[h]\rangle$ on the stack, then $h' < j$. If, when we reach position $i \geq h$, the pair $\langle j, \mathsf{lcp}[h]\rangle$ is in the stack this means that all positions $k$ between $j$ and $i$ we have $\mathsf{lcp}[k] \geq \mathsf{lcp}[h]$. To maintain this invariant, when we reach position $i$, if the current top pair $\langle j, \mathsf{lcp}[h]\rangle$ has $\mathsf{lcp}[h] < \mathsf{lcp}[i]$, then $\langle i, \mathsf{lcp}[i]\rangle$ is pushed on top of the stack. Otherwise, all pairs $\langle j, \mathsf{lcp}[h]\rangle$ with $\mathsf{lcp}[h] \geq \mathsf{lcp}[i]$ are popped from the stack; if $\hat{\jmath}$ is the index of the last pair popped from the stack, pair $\langle \hat{\jmath}, \mathsf{lcp}[i]\rangle$ is pushed on the stack. The rationale for the latter addition is that for all $\hat{\jmath} \leq j \leq i$ it is $\mathsf{lcp}[j] \geq \mathsf{lcp}[i]$ and therefore the prefix of length $\mathsf{lcp}[i]$ of $\mathsf{s}[\mathsf{sa}[j], n]$ is the same as the prefix of the same length of $\mathsf{s}[\mathsf{sa}[i], n]$. It is not difficult to prove that for each stack entry $\langle j, \mathsf{lcp}[h]\rangle$, it is $\mathsf{lcp}[j-1] < \mathsf{lcp}[h]$.

If entry $\langle j+1, \mathsf{lcp}[h]\rangle$ is removed from the stack at iteration $i+1$, by the above discussion $\mathsf{lcp}[j] < \mathsf{lcp}[h]$; $\mathsf{lcp}[i+1] < \mathsf{lcp}[h]$ (because $\langle j+1, \mathsf{lcp}[h]\rangle$ is being removed), and for $k = j+1, \ldots, i$ $\mathsf{lcp}[k] \geq \mathsf{lcp}[h]$. To ensure that we have found a Type 1 maximal repeat we only need to check that $\mathsf{bwt}[j, i]$ contains at least two distinct characters. To efficiently check this latter condition, for each stack entry $\langle j, \mathsf{lcp}[h]\rangle$ we maintain a bitvector $b_j$ of size $\sigma$ keeping track of the distinct characters in the array $\mathsf{bwt}$ from position $j-1$ to the next stack entry, or to the last seen position for the entry at the top of the stack. When $\langle j, \mathsf{lcp}[h]\rangle$ is popped from the stack its bitvector is or-ed to the previous stack entry in constant time; if $\langle j, \mathsf{lcp}[h]\rangle$ is popped from the stack and immediately replaced with $\langle j, \mathsf{lcp}[i]\rangle$ its bitvector survives as it is (essentially because it is associated with an index, not with a stack entry). Clearly, maintaining the bitvector does not increase the asymptotic cost of the algorithm.

To find Type 2 maximal repeats, we are interested in consecutive LCP entries $\mathsf{lcp}[j], \mathsf{lcp}[j+1], \ldots, \mathsf{lcp}[i], \mathsf{lcp}[i+1]$, such that $\mathsf{lcp}[j] < \mathsf{lcp}[j+1] = \mathsf{lcp}[j+2] = \cdots = \mathsf{lcp}[i] > \mathsf{lcp}[i+1]$. Indeed, this ensures that for $h = j, \ldots, i$ all suffixes $\mathsf{s}[\mathsf{sa}[h], n]$ are prefixed by the same string $\alpha$ of length $\mathsf{lcp}[j+1]$ and every extension $\alpha c$ occurs at most once. If this is the case, then $\alpha$ is a Type 2 maximal repeat if all characters in $\mathsf{bwt}[j, i]$ are distinct since this ensures that also every extension $c\alpha$ occurs at most once. In order to detect this situation, as we scan the $\mathsf{lcp}$ array we maintain a candidate pair $\langle j+1, \mathsf{lcp}[j+1]\rangle$ such that $j+1$ is the largest index seen so far for which $\mathsf{lcp}[j] < \mathsf{lcp}[j+1]$. When we establish a candidate at $j+1$ as above, we init a bitvector $b$ marking entries $\mathsf{bwt}[j]$ and $\mathsf{bwt}[j+1]$. As long as the following values $\mathsf{lcp}[j+2], \mathsf{lcp}[j+3], \ldots$ are equal to $\mathsf{lcp}[j+1]$ we go on updating $b$ and if the same position is marked twice we discard $\langle j+1, \mathsf{lcp}[j+1]\rangle$. If we reach an index $i+1$ such that $\mathsf{lcp}[i+1] > \mathsf{lcp}[j+1]$, we update the candidate and reinitialize $b$. If we reach $i+1$ such that $\mathsf{lcp}[i+1] < \mathsf{lcp}[j+1]$ and $\langle j+1, \mathsf{lcp}[j+1]\rangle$ has not been discarded, then a repeat of Type 2 (with $i-j+1$ repetitions) has been located.

Note that when our algorithms discover Type 1 or Type 2 maximal repeats, we know the repeat length and the number of occurrences, so one can easily filter out non-interesting repeats (too short or too frequent). In some applications, for example the MUMmer tool [28], one is interested in repeats that occur in at least $r$ distinct sequences, maybe exactly once for each sequence. Since for these applications the number of distinct sequences is relatively small, we can handle this requirements by simply scanning the $\mathsf{da}$ array simultaneously with the $\mathsf{lcp}$ and $\mathsf{bwt}$ arrays and keeping track of the sequences associated to a maximal repeat using a bitvector (or a union-find structure) as we do with characters in the $\mathsf{bwt}$.

## 5.2 All pairs suffix-prefix overlaps

In this problem we want to compute, for each pair of sequences $\mathsf{s}_i$ $\mathsf{s}_j$, the longest overlap between a suffix of $\mathsf{s}_i$ and a prefix of $\mathsf{s}_j$. Our solution follows closely the one in [33] which in turn was inspired by an earlier Suffix-tree based algorithm [16]. The algorithm in [33] solves the problem using a Generalized Enhanced Suffix array (consisting of the arrays $\mathsf{sa}$, $\mathsf{lcp}$, and $\mathsf{da}$) in $\mathcal{O}(n + m^2)$ time, which is optimal since there are $m^2$ overlaps. However, for large collections it is natural to consider the problem of reporting only the overlaps larger than a given threshold $\tau$ still spending constant time per reported overlap. Our algorithm solves this more challenging problem.

In the following we say that the suffix starting at $\mathsf{sa}[i]$ is *special* iff $\mathsf{s}[\mathsf{sa}[i] + \mathsf{lcp}[i+1]] = \$$ or, in other words, if the suffix starting at $\mathsf{sa}[i]$ is a substring of the suffix starting at $\mathsf{sa}[i+1]$ (not considering the end-marker \$). For example, in Fig. 1 (right) the suffixes $\mathsf{ab}\$_0$, $\mathsf{abc}\$_1$, $\mathsf{abcab}\$_0$ are all special. We can modify Phase 2 of our algorithm so that it outputs also a bit array $\mathsf{xlcp}$ such that $\mathsf{xlcp}[i] = \mathbf{1}$ iff the suffix starting at $\mathsf{sa}[i]$ is special. In the full paper we will formally prove that this modification does not increase the asymptotic cost and requires only $2n$ bits of disk working space.

Our algorithm consists of a sequential scan of the arrays $\mathsf{bwt}$, $\mathsf{lcp}$, and $\mathsf{da}$, and $\mathsf{xlcp}$. We maintain $m$ distinct stacks, $\mathsf{stack}[1], \ldots, \mathsf{stack}[m]$, one for each input sequence; $\mathsf{stack}[k]$ stores only *special* suffixes belonging to sequence $k$. When the scanning reaches position $j$, we store the pair $\langle j, \mathsf{lcp}[j+1]\rangle$ in $\mathsf{stack}[\mathsf{da}[j]]$ if and only if $\mathsf{xlcp}[j] = \mathbf{1}$ and $\mathsf{lcp}[j+1] > \tau$. During the scanning we maintain the invariant that for all stack entries $\langle j, \mathsf{lcp}[j+1]\rangle$, $\mathsf{lcp}[j+1]$ is the length of longest common prefix between $\mathsf{s}[\mathsf{sa}[j], n]$ and $\mathsf{s}[\mathsf{sa}[i], n]$, where $i$ is the next position to be scanned. To this end, when we reach position $i$ we remove all entries $\langle j, \mathsf{lcp}[j+1]\rangle$ such that $\mathsf{lcp}[j+1] > \mathsf{lcp}[i+1]$. To do this spending constant time for removed entry requires some additional machinery: We maintain an array of lists $\mathsf{top}$ such that $\mathsf{top}[\ell]$ contains the indexes $k$ for which the entry at the top of $\mathsf{stack}[k]$ has LCP component equal to $\ell$ (this array is a stripped down version of [33]'s $\mathsf{list}$). In addition, we maintain an additional stack $\mathsf{lcpStack}$ containing, in increasing order, the values $\ell$ such that some $\mathsf{stack}[k]$ contains an entry with LCP component equal to $\ell$.

At iteration $i$, we use $\mathsf{lcpStack}$ and the lists in $\mathsf{top}[\cdot]$ to reach all $\mathsf{stack}[k]$ containing entries with LCP component greater than $\mathsf{lcp}[i+1]$ and we remove them. After the removal, we update $\mathsf{top}[\ell]$ where $\ell$ is the LCP value now at the top of $\mathsf{stack}[k]$. Finally, if $\mathsf{xlcp}[i] = \mathbf{1}$ and $\mathsf{lcp}[i+1] > \tau$, we add $\langle i, \mathsf{lcp}[i+1]\rangle$ to $\mathsf{stack}[\mathsf{da}[i]]$; this requires that we also add $\mathsf{da}[i]$ to $\mathsf{top}[\mathsf{lcp}[i+1]]$, and that we remove $\mathsf{da}[i]$ from the list $\mathsf{top}[\ell]$ where $\ell$ is the previous top LCP value in $\mathsf{stack}[\mathsf{da}[i]]$ (to do this we need to maintain for each element at the top of the stack a pointer to its corresponding $\mathsf{da}$ entry in $\mathsf{top}$). Since we perform a constant number of operations per entry, maintaining the above data structures takes $\mathcal{O}(n)$ time overall.

The computation of the overlaps is done as in [33]. When the scanning reaches position $i$, we check whether $\mathsf{bwt}[i] = \$$. If this is the case, then $\mathsf{s}[\mathsf{sa}[i], n]$ is prefixed by the whole sequence $\mathsf{s}_{\mathsf{da}[i]}$, hence the longest overlap between a prefix of $\mathsf{s}_{\mathsf{da}[i]}$ and a suffix of $\mathsf{s}_k$ is given by the element currently at the top of $\mathsf{stack}[k]$, since by construction these stacks only contain special suffixes whose overlap with $\mathsf{s}[\mathsf{sa}[i], n]$ is larger than $\tau$. To spend time proportional to the number of reported overlaps, instead of accessing all stacks we access only those which are non-empty. This requires that we maintain an additional list containing all values $\ell$ such that top$[\ell]$ is non-empty. For each entry $\ell$ in this list, top$[\ell]$ gives us the id of the sequences with a suffix-prefix overlap with da$[i]$ of length $\ell$. As in [33], we have to handle differently the case in which the whole $\mathsf{s}_{\mathsf{da}[i]}$ is a suffix of another sequence, but this can be done without increasing the overall complexity. Since we spend constant time for reported overlap, the overall cost of the algorithm, in addition to the scanning of the bwt/lcp/xlcp/da arrays, is $\mathcal{O}(n + E_\tau)$, where $E_\tau$ is the number of suffix-prefix overlaps greater than $\tau$.

## 5.3 Construction of succinct de Bruijn graphs

A recent remarkable application of compressed data structures is the design of efficiently navigable succinct representations of de Bruijn graphs [3, 7, 8]. Formally, a de Bruijn graph for a collection of strings consists of a set of vertices representing the distinct $k$-mers appearing in the collection, with a directed edge $(u, v)$ iff there exists a $(k+1)$-mer $\alpha$ in the collection such that $\alpha[1, k]$ is the $k$-mer associated to $u$ and $\alpha[2, k+1]$ is the $k$-mer associated to $v$.

The starting point of all de Bruijn graphs succinct representation is the BOSS representation [8], so called from the authors' initials. For simplicity we now describe the BOSS representation of a $k$-order de Bruijn graph using the lexicographic order of $k$-mers, instead of the co-lexicographic order as in [8], which means we are building the graph with the direction of the arcs reversed. This is not a limitation since arcs can be traversed in both directions (or we can apply our construction to the input sequences reversed).

Consider the $N$ $k$-mers appearing in the collection sorted in lexicographic order. For each $k$-mer $\alpha_i$ consider the array $C_i$ of distinct characters $c \in \Sigma \cup \{\$\}$ such that $c\alpha_i$ appears in the collection. The concatenation $W = C_1 C_2 \cdots C_N$ is the first component of the BOSS representation. The second component is a binary array *last*, with $|last| = |W|$, such that $last[j] = \mathbf{1}$ iff $W[j]$ is the last entry of some array $C_i$. Clearly, there is a bijection between entries in $W$ and graph edges; in the array *last* each sequence $\mathbf{0}^i\mathbf{1}$ $(i \geq 0)$ corresponds to the outgoing edges of a single vertex with outdegree $i+1$. Finally, the third component is a binary array $W^-$, with $|W^-| = |W|$, such that $W^-[j] = \mathbf{1}$ iff $W[j]$ comes from the array $C_i$, where $\alpha_i$ is the lexicographically smallest $k$-mer prefixed by $\alpha_i[1, k-1]$ and preceded by $W[j]$ in the collection. Informally, this means that $\alpha_i$ is the lexicographically smallest $k$-mer with an outgoing edge reaching $W[j]\alpha_i[1, k-1]$. Note that the number of $\mathbf{1}$'s in *last* and $W^-$ is exactly $N$, i.e. the number of nodes in the de Bruijn graph.

We now show how to compute $W$, *last* and $W^-$ by a sequential scan of the bwt and lcp array. The crucial observation is that the suffix array range prefixed by the same $k$-mer $\alpha_i$ is identified by a range $[b_i, e_i]$ of LCP values satisfying $\mathsf{lcp}[b_i] < k$, $\mathsf{lcp}[\ell] \geq k$ for $\ell = b_i + 1, \ldots, e_i$ and $\mathsf{lcp}[e_i + 1] < k$. Since $k$-mers are scanned in lexicographic order, by keeping track of the corresponding characters in the array $\mathsf{bwt}[b_i, e_i]$ we can build the array $C_i$ and consequently $W$ and *last*. To compute $W^-$ we simply need to keep track also of suffix array ranges corresponding to $(k-1)$-mers. Every time we set an entry $W[j] = c$ we set $W^-[j] = \mathbf{1}$ iff this is the first occurrence of $c$ in the range corresponding to the current $(k-1)$-mers.

If, in addition to the bwt and lcp arrays, we also scan the da array, then we can keep track of which sequences contain any given graph edge and therefore obtain a succinct representation of the colored de Bruijn graph [29]. Finally, we observe that if our only objective is to build the $k$-order de Bruijn graph, then we can stop the phase 2 of our algorithm after the $k$-th iteration. Indeed, we do not need to compute the exact values of LCP entries greater than $k$, and also we do not need the exact BWT but only the BWT characters sorted by their length $k$ context.

## References

1 Markus J. Bauer, Anthony J. Cox, and Giovanna Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theor. Comput. Sci.*, 483:134–148, 2013.

2 Djamal Belazzougui. Linear time construction of compressed text indices in compact space. In *STOC*, pages 148–193. ACM, 2014.

3 Djamal Belazzougui, Travis Gagie, Veli Mäkinen, Marco Previtali, and Simon J. Puglisi. Bidirectional variable-order de Bruijn graphs. In *LATIN*, volume 9644 of *Lecture Notes in Computer Science*, pages 164–178. Springer, 2016.

4 Paola Bonizzoni, Gianluca Della Vedova, Yuri Pirola, Marco Previtali, and Raffaella Rizzi. Computing the BWT and LCP array of a set of strings in external memory. *CoRR*, abs/1705.07756, 2017.

5 Paola Bonizzoni, Gianluca Della Vedova, Yuri Pirola, Marco Previtali, and Raffaella Rizzi. An external-memory algorithm for string graph construction. *Algorithmica*, 78(2):394–424, 2017. doi:10.1007/s00453-016-0165-4.

6 Paola Bonizzoni, Gianluca Della Vedova, Yuri Pirola, Marco Previtali, and Raffaella Rizzi. Divide and conquer computation of the multi-string BWT and LCP array. In *Proc. Computability in Europe (CiE)*, 2018. To appear.

7 Christina Boucher, Alexander Bowe, Travis Gagie, Simon J. Puglisi, and Kunihiko Sadakane. Variable-order de Bruijn graphs. In *DCC*, pages 383–392. IEEE, 2015.

8 Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn graphs. In *WABI*, volume 7534 of *Lecture Notes in Computer Science*, pages 225–235. Springer, 2012.

9 S. Burkhardt and J. Kärkkäinen. Fast lightweight suffix array construction and checking. In *Proc. 14th Symposium on Combinatorial Pattern Matching (CPM '03)*, pages 55–69. Springer-Verlag LNCS n. 2676, 2003.

10 Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital SRC Research Report, 1994.

11 Anthony J. Cox, Fabio Garofalo, Giovanna Rosone, and Marinella Sciortino. Lightweight LCP construction for very large collections of strings. *J. Discrete Algorithms*, 37, 2016.

12 Lavinia Egidi and Giovanni Manzini. Lightweight BWT and LCP merging via the Gap algorithm. In *SPIRE*, volume 10508 of *Lecture Notes in Computer Science*, pages 176–190. Springer, 2017.

13 P. Ferragina, T. Gagie, and G. Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 2011.

14 Simon Gog and Enno Ohlebusch. Compressed suffix trees: Efficient computation and storage of LCP-values. *ACM Journal of Experimental Algorithmics*, 18, 2013. doi:10.1145/2444016.2461327.

15 D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

16 Dan Gusfield, Gad M. Landau, and Baruch Schieber. An efficient algorithm for the all pairs suffix-prefix problem. *Inf. Process. Lett.*, 41(4):181–185, 1992.

17 James Holt and Leonard McMillan. Merging of multi-string BWTs with applications. *Bioinformatics*, 30(24):3524–3531, 2014.

18 Juha Kärkkäinen and Dominik Kempa. LCP array construction in external memory. *ACM Journal of Experimental Algorithmics*, 21(1):1.7:1–1.7:22, 2016.

19 Juha Kärkkäinen and Dominik Kempa. Engineering a lightweight external memory suffix array construction algorithm. *Mathematics in Computer Science*, 11(2):137–149, 2017.

20 D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, second edition, 1998.

**21** M. Oguzhan Külekci, Jeffrey Scott Vitter, and Bojian Xu. Efficient maximal repeat finding using the burrows-wheeler transform and wavelet tree. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 9(2):421–429, 2012.

**22** Felipe A. Louza, Simon Gog, and Guilherme P. Telles. Inducing enhanced suffix arrays for string collections. *Theor. Comput. Sci.*, 678:22–39, 2017.

**23** Felipe A. Louza, Guilherme P. Telles, Steve Hoffmann, and Cristina D. A. Ciferri. Generalized enhanced suffix array construction in external memory. *Algorithms for Molecular Biology*, 12(1):26:1–26:16, 2017.

**24** Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing*. Cambridge University Press, 2015.

**25** Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows-Wheeler transform. *Theor. Comput. Sci.*, 387(3):298–312, 2007.

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

**27** G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. In *Proc. 10th European Symposium on Algorithms (ESA)*, pages 698–710. Springer Verlag LNCS n. 2461, 2002.

**28** Guillaume Marçais, Arthur L. Delcher, Adam M. Phillippy, Rachel Coston, Steven L. Salzberg, and Aleksey V. Zimin. Mummer4: A fast and versatile genome alignment system. *PLoS Computational Biology*, 14(1), 2018.

**29** Martin D. Muggli, Alexander Bowe, Noelle R. Noyes, Paul S. Morley, Keith E. Belk, Robert Raymond, Travis Gagie, Simon J. Puglisi, and Christina Boucher. Succinct colored de Bruijn graphs. *Bioinformatics*, 33(20):3181–3187, 2017.

**30** J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In *SODA*, pages 408–424. SIAM, 2017.

**31** G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007.

**32** Ge Nong. Practical linear-time $O(1)$-workspace suffix sorting for constant alphabets. *ACM Trans. Inf. Syst.*, 31(3):15, 2013.

**33** Enno Ohlebusch and Simon Gog. Efficient algorithms for the all-pairs suffix-prefix problem and the all-pairs substring-prefix problem. *Inf. Process. Lett.*, 110(3):123–128, 2010.

**34** Enno Ohlebusch, Simon Gog, and Adrian Kügel. Computing matching statistics and maximal exact matches on compressed full-text indexes. In *SPIRE*, volume 6393 of *Lecture Notes in Computer Science*, pages 347–358. Springer, 2010.

**35** William H. A. Tustumi, Simon Gog, Guilherme P. Telles, and Felipe A. Louza. An improved algorithm for the all-pairs suffix-prefix problem. *J. Discrete Algorithms*, 37:34–43, 2016.