Lightweight BWT and LCP Merging via the Gap Algorithm

Lavinia Egidi[1(✉)] and Giovanni Manzini[1,2(✉)]

[1] Computer Science Institute, DiSIT University of Eastern Piedmont, Viale Teresa Michel, 11, 15100 Alessandria, Italy
{lavinia.egidi,giovanni.manzini}@uniupo.it

[2] Institute of Informatics and Telematics CNR, Via Moruzzi, 1, 56124 Pisa, Italy

Abstract. Recently, Holt and McMillan [Bioinformatics 2014, ACM-BCB 2014] have proposed a simple and elegant algorithm to merge the Burrows-Wheeler transforms of a collection of strings. In this paper we show that their algorithm can be improved so that, in addition to the BWTs, it also merges the Longest Common Prefix (LCP) arrays. Because of its small memory footprint this new algorithm can be used for the final merge of BWT and LCP arrays computed by a faster but memory intensive construction algorithm.

Keywords: Document collections · String indexing · Data compression

1 Introduction and Related Works

The Burrows Wheeler transform (BWT) is a fundamental component of many compressed indices and it is often complemented by the Longest Common Prefix (LCP) array and a sampling of the Suffix Array [9,21]. Because of the sheer size of the data involved, the construction of such data structures is a challenging problem in itself. 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 [4,19,20]. When even lightweight algorithms do not fit in RAM, one has to resort to external memory construction algorithms (see [5,7,13,17] and references therein).

Many construction algorithms are designed for the case in which the input consists of a single sequence; yet in many applications the data to be indexed consist of a collection of distinct items: documents, web pages, NGS reads, proteins, *etc.*. One can concatenate such items using (distinct) end-of-file separators and index the resulting sequence. However, using distinct separators is possible only

Partially supported by the University of Eastern Piedmont projects KITE and *Behavioural Types for Dependability Analysis with Bayesian Networks*, PRIN grant 201534HNXC, and INdAM-GNCS Project *Efficient algorithms and techniques for the organization, management and analysis of biological Big Data*.

for small collections and from the algorithmic point of view it makes no sense to "forget" that the input consists of distinct items: this additional information should be exploited to run faster.

Recently, Holt and McMillan [10,11] have presented a new approach for computing the BWT of a collection of sequences based on the concept of merging: first the BWTs of the individual sequences are computed (by any single-string BWT algorithm) and then they are merged, possibly in multiple rounds as in the standard mergesort algorithm. The idea of BWT-merging is not new [6,23] but Holt and McMillan's merging algorithm is simpler than the previous approaches. For a constant size alphabet their algorithm merges the BWTs of two sequences t_0, t_1 in $\mathcal{O}(n \cdot \mathsf{avelcp}_{01})$ time where $n = |\mathsf{t}_0| + |\mathsf{t}_1|$ and avelcp_{01} is the average length of the longest common prefix between suffixes of t_0 and t_1. The average length of the longest common prefix is $\mathcal{O}(n)$ in the worst case but $\mathcal{O}(\log n)$ for random strings and for many real world datasets [14]. Note that even when $\mathsf{avelcp}_{01} = \mathcal{O}(\log n)$ the algorithm is not optimal since BWT merging can be done in linear time if there are no constraints on the space usage.

In this paper we show that the H&M (Holt and McMillan) merging algorithm can be modified so that, in addition to the BWTs, it merges the LCP arrays as well. The new algorithm, called Gap because of how it operates, has the same asymptotic cost as H&M and uses additional space only for storing its additional output, i.e. the LCP values. In our implementation, the Gap algorithm uses only $\approx$ 1.5 bytes per symbol of workspace in addition to the input and the output, making it interesting when the overall size of the collection is close to the available RAM.

Our contribution in context. For a collection of documents of total size n over a constant alphabet, the BWT and LCP arrays, as well as many compressed indices, can be computed in $\mathcal{O}(n)$ time by first computing the Suffix Array (SA) of the collection. The construction of the suffix array is a well studied problem and there exist time/space optimal algorithms that work well in practice. The problem with this approach is that the SA takes $n \log n$ bits of space while a compressed index takes $\mathcal{O}(n)$ bits. Hence, going through the SA we can only build indices much smaller than the available RAM. This implies that, in practice, either we build multiple "small" indices, which must be queried independently, or we use a larger machine for the construction of the index. Note that the construction of compressed indices in linear time and $\mathcal{O}(n)$ bits of space is a challenging and active area of research, see [1] and references therein, but at the moment it has produced no practical algorithms.

Given this state of affairs, we propose the following practical approach for the construction of the BWT and LCP arrays of a collection of documents. We split the input collection into subcollections $C_1, \ldots, C_k$ of roughly equal size which are sufficiently small so that we can compute the BWT and LCP arrays via the SA. Then, we merge all the BWTs and LCPs using the Gap algorithm described in this paper. Since space is the main bottleneck, to compute the SA of the subcollections we use the recently proposed gSACA-K algorithm by Louza et al. [15,16] which runs in linear time, is extremely fast in practice, and uses only 2 KB in addition to the space of the input and the output (gSACA-K is based on the SACA-K algorithm from [22]). As we will see, this approach allows us to fully exploit all the available RAM and to take advantage of the optimal and highly engineered gSACA-K algorithm to do most of the job.

Among the construction algorithms proposed in the literature, the one most similar to our approach is the one described by Sirén in [24] where a compressed index is maintained in RAM and new collections are incrementally merged to it. The two approaches share the idea that building index should not require a specialized machine with a lot of RAM. The approach in [24] is specific for that particular compressed index (which doesn't use the LCP array), while ours, providing the plain BWT and LCP arrays, can be more easily adapted to build different flavors of compressed indices.

Another related result is the algorithm proposed in [2,3] which computes (from scratch) the multi-string BWT and LCP in *external memory*. Given a collection of m strings of the same length k, the algorithm first computes the BWT of all length-ℓ suffixes for $\ell = 1, 2, \ldots, k$ and then, using an approach inspired by the H&M algorithm, merges them to obtain the multi-string BWT and LCP arrays. The algorithm accesses disk data sequentially and the reported I/O volume is $\mathcal{O}(mk\,\mathsf{maxlcp})$ where maxlcp is the maximum of the length of all common prefixes (for the same input the I/O volume of the algorithm in [5] is $\mathcal{O}(mk^2)$). Although our Gap algorithm is designed only to merge BWTs and LCPs in internal memory, it also accesses its main data structures sequentially. This feature suggests that it could be engineered to work in external memory as well. Compared to [2,3] an external memory version of Gap would have the advantages of supporting also strings of different lengths, and of exploiting any available RAM to do some of the work with the highly efficient, indeed optimal, internal memory algorithm gSACA-K. We plan to pursue this line of research in a future work.

2 Notation

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

The *suffix array* $\mathsf{sa}[1,n]$ associated to t is the permutation of $[1,n]$ giving the lexicographic order of t's suffixes, that is, for $i = 1,\ldots,n-1$, $\mathsf{t}[\mathsf{sa}[i],n] \prec \mathsf{t}[\mathsf{sa}[i+1],n]$. The *longest common prefix* array $\mathsf{lcp}[1,n+1]$ is defined for $i = 2,\ldots,n$ by

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

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

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

lcp	bwt	*context*
-1	c	•
0	•	aabcabc•
1	c	abc•
3	a	abcabc•
0	a	bc•
2	a	bcabc•
0	b	c•
1	b	cabc•
-1		

id	lcp_{01}	bwt_{01}	*context*
0	-1	b	$
1	0	c	•
1	0	•	aabcabc•
0	1	c	ab$
1	2	c	abc•
0	3	$	abcab$
1	5	a	abcabc•
0	0	a	b$
1	1	a	bc•
0	2	a	bcab$
1	4	a	bcabc•
1	0	b	c•
0	1	b	cab$
1	3	b	cabc•
	-1		

Fig. 1. LCP array and BWT for $\mathsf{t}_0 = \mathsf{abcab\$}$ and $\mathsf{t}_1 = \mathsf{aabcabc}\bullet$, and multi-string BWT and corresponding LCP array for the same strings. Column id shows, for each entry of $\mathsf{bwt}_{01} = \mathsf{bc{\bullet}cc\$aaaabbb}$ whether it comes from t_0 or t_1.

The *Burrows-Wheeler transform* $\mathsf{bwt}[1, n]$ of t is defined by

$$\mathsf{bwt}[i] = \begin{cases} \mathsf{t}[n] & \text{if } \mathsf{sa}[i] = 1 \\ \mathsf{t}[\mathsf{sa}[i] - 1] & \text{if } \mathsf{sa}[i] > 1. \end{cases}$$

$\mathsf{bwt}[1, n]$ is the permutation of t in which the position of $\mathsf{t}[j]$ coincides with the lexicographic rank of $\mathsf{t}[j+1, n]$ (or of $\mathsf{t}[1, n]$ if $j = n$) in the suffix array. We call such string the *context* of $\mathsf{t}[j]$. See Fig. 1 for an example.

The longest common prefix (LCP) array, and Burrows-Wheeler transform (BWT) can be generalized to the case of multiple strings [5,18]. Let $\mathsf{t}_0[1, n_0]$ and $\mathsf{t}_1[1, n_1]$ be such that $\mathsf{t}_0[n_0] = \$_0$ and $\mathsf{t}_1[n_1] = \$_1$ where $\$_0 < \$_1$ are two symbols not appearing elsewhere in t_0 and t_1 and smaller than any other symbol. Let $\mathsf{sa}_{01}[1, n_0 + n_1]$ denote the suffix array of the concatenation $\mathsf{t}_0\mathsf{t}_1$. The *multi-string* BWT of t_0 and t_1, denoted by $\mathsf{bwt}_{01}[1, n_0 + n_1]$, is defined by

$$\mathsf{bwt}_{01}[i] = \begin{cases} \mathsf{t}_0[n_0] & \text{if } \mathsf{sa}_{01}[i] = 1 \\ \mathsf{t}_0[\mathsf{sa}_{01}[i] - 1] & \text{if } 1 < \mathsf{sa}_{01}[i] \le n_0 \\ \mathsf{t}_1[n_1] & \text{if } \mathsf{sa}_{01}[i] = n_0 + 1 \\ \mathsf{t}_1[\mathsf{sa}_{01}[i] - n_0 - 1] & \text{if } n_0 + 1 < \mathsf{sa}_{01}[i]. \end{cases}$$

In other words, $\mathsf{bwt}_{01}[i]$ is the symbol preceding the i-th lexicographically larger suffix, with the exception that if $\mathsf{sa}_{01}[i] = 1$ then $\mathsf{bwt}_{01}[i] = \$_0$ and if $\mathsf{sa}_{01}[i] = n_0 + 1$ then $\mathsf{bwt}_{01}[i] = \$_1$. Hence, $\mathsf{bwt}_{01}[i]$ always comes from the string (t_0 or t_1) containing the i-th largest suffix (see again Fig. 1). The above notion of multi-string BWT can be immediately generalized to define $\mathsf{bwt}_{1\cdots k}$ for a family of distinct strings $\mathsf{t}_1, \mathsf{t}_2, \ldots, \mathsf{t}_k$. Essentially $\mathsf{bwt}_{1\cdots k}$ is a permutation of the symbols in $\mathsf{t}_1, \ldots, \mathsf{t}_k$ such that the position in $\mathsf{bwt}_{1\cdots k}$ of $\mathsf{t}_i[j]$ is given by the lexicographic rank of its context $\mathsf{t}_i[j+1, n_i]$ (or $\mathsf{t}_i[1, n_i]$ if $j = n_i$).

Given the concatenation $\mathsf{t}_0\mathsf{t}_1$ and its suffix array $\mathsf{sa}_{01}[1, n_0 + n_1]$, we consider the corresponding LCP array $\mathsf{lcp}_{01}[1, n_0 + n_1 + 1]$ defined as in (1) (see again Fig. 1). Note that, for $i = 2, \ldots, n_0 + n_1$, $\mathsf{lcp}_{01}[i]$ gives the length of the longest common prefix between the contexts of $\mathsf{bwt}_{01}[i]$ and $\mathsf{bwt}_{01}[i-1]$. This definition can be immediately generalized to a family of k strings to define the LCP array $\mathsf{lcp}_{12\cdots k}$ associated to the multi-string BWT $\mathsf{bwt}_{12\cdots k}$.

3 The H&M Algorithm Revisited

In [11] Holt and McMillan introduced a simple and elegant algorithm, we call it the H&M algorithm, to merge multi-string BWTs as defined above.

Given $\mathsf{bwt}_{1\cdots k}$ and $\mathsf{bwt}_{k+1\, k+2\cdots h}$ the algorithm computes $\mathsf{bwt}_{1\cdots h}$. The computation does not explicitly need $\mathsf{t}_1, \ldots, \mathsf{t}_h$ but only the (multi-string) BWTs to be merged. For simplicity of notation we describe the algorithm assuming we are merging two single-string BWTs $\mathsf{bwt}_0 = \mathsf{bwt}(\mathsf{t}_0)$ and $\mathsf{bwt}_1 = \mathsf{bwt}(\mathsf{t}_1)$; the algorithm does not change in the general case where the input are multi-string BWTs. Note also that the algorithm can be easily adapted to merge more than two (multi-string) BWTs at the same time.

Computing bwt_{01} amounts to sorting the symbols of bwt_0 and bwt_1 according to the lexicographic order of their contexts, where the context of symbol $\mathsf{bwt}_0[i]$ (resp. $\mathsf{bwt}_1[i]$) is $\mathsf{t}_0[\mathsf{sa}_0[i], n_0]$ (resp. $\mathsf{t}_1[\mathsf{sa}_1[i], n_1]$). By construction, the symbols in bwt_0 and bwt_1 are already sorted by context, hence to compute bwt_{01} we only need to merge bwt_0 and bwt_1 without changing the relative order of the symbols within the two input sequences.

The H&M algorithm works in successive phases. After the h-th phase the entries of bwt_0 and bwt_1 are sorted on the basis of the first h symbols of their context. More formally, the output of the h-th phase is a binary vector $Z^{(h)}$ containing $n_0 = |\mathsf{t}_0|$ $\mathbf{0}$'s and $n_1 = |\mathsf{t}_1|$ $\mathbf{1}$'s and such that the following property holds.

Property 1. For $i = 1, \ldots, n_0$ and $j = 1, \ldots n_1$ the i-th $\mathbf{0}$ precedes the j-th $\mathbf{1}$ in $Z^{(h)}$ if and only if

$$\mathsf{t}_0[\mathsf{sa}_0[i], \mathsf{sa}_0[i] + h - 1] \preceq \mathsf{t}_1[\mathsf{sa}_1[j], \mathsf{sa}_1[j] + h - 1] \tag{2}$$

(recall that according to our notation if $\mathsf{sa}_0[i] + h - 1 > n_0$ then $\mathsf{t}_0[\mathsf{sa}_0[i], \mathsf{sa}_0[i] + h - 1]$ coincides with $\mathsf{t}_0[\mathsf{sa}_0[i], n_0]$, and similarly for t_1). $\square$

Following Property 1 we identify the i-th $\mathbf{0}$ in $Z^{(h)}$ with $\mathsf{bwt}_0[i]$ and the j-th $\mathbf{1}$ in $Z^{(h)}$ with $\mathsf{bwt}_1[j]$ so that to $Z^{(h)}$ corresponds to a permutation of bwt_{01}. Property 1 is equivalent to state that we can logically partition $Z^{(h)}$ into $b(h) + 1$ blocks

$$Z^{(h)}[1, \ell_1], Z^{(h)}[\ell_1 + 1, \ell_2], \ldots, Z^{(h)}[\ell_{b(h)} + 1, n_0 + n_1] \tag{3}$$

such that each block corresponds to a set of bwt_{01} symbols whose contexts are prefixed by the same length-h string (the symbols with a context of length less than h are contained in singleton blocks). Within each block the symbols of bwt_0 precede those of bwt_1, 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 $k>j$.

The H&M algorithm initially sets $Z^{(0)} = \mathbf{0}^{n_0}\mathbf{1}^{n_1}$: since the context of every bwt_{01} symbol is prefixed by the same length-0 string (the empty string), there is a single block containing all bwt_{01} symbols. At phase h the algorithm computes $Z^{(h+1)}$ from $Z^{(h)}$ using the procedure in Fig. 2. For completeness we report in the Appendix the proof of the following lemma which is a restatement of Lemma 3.2 in [11] using our notation.

```
 1: Initialize array $F[1,\sigma]$
 2: $k_0 \leftarrow 1$; $k_1 \leftarrow 1$                    ▷ Init counters for $\mathsf{bwt}_0$ and $\mathsf{bwt}_1$
 3: for $k \leftarrow 1$ to $n_0+n_1$ do
 4:     $b \leftarrow Z^{(h-1)}[k]$                              ▷ Read bit $b$ from $Z^{(h-1)}$
 5:     if $b = 0$ then              ▷ Get symbol from $\mathsf{bwt}_0$ or $\mathsf{bwt}_1$ according to $b$
 6:         $c \leftarrow \mathsf{bwt}_0[k_0{+}{+}]$
 7:     else
 8:         $c \leftarrow \mathsf{bwt}_1[k_1{+}{+}]$
 9:     end if
10:     $j \leftarrow F[c]{+}{+}$                 ▷ Get destination for $b$ according to symbol $c$
11:     $Z^{(h)}[j] \leftarrow b$                                  ▷ Copy bit $b$ to $Z^{(h)}$
12: end for
```

Fig. 2. Main loop of algorithm H&M for computing $Z^{(h)}$ given $Z^{(h-1)}$. Array F is initialized so that $F[c]$ contains the number of occurrences of symbols smaller than c in bwt_0 and bwt_1 plus one. Hence, the bits stored in $Z^{(h)}$ immediately after reading symbol c are stored in positions from $F[c]$ to $F[c+1]-1$ of $Z^{(h)}$.

Lemma 2. *For $h = 0, 1, 2, \ldots$ the bit vector $Z^{(h)}$ satisfies Property 1.* □

We now show that with a simple modification to the H&M algorithm it is possible to compute, in addition to bwt_{01} also the LCP array lcp_{01} defined in Sect. 2. Our strategy consists in keeping explicit track of the logical blocks we have defined for $Z^{(h)}$ and represented in (3). We maintain an integer array $B[1, n_0+n_1+1]$ such that at the end of phase h it is $B[i] \neq 0$ if and only if a block of $Z^{(h)}$ starts at position i. The use of such integer array is shown in Fig. 3. Note that: (i) initially we set $B = 1\,0^{n_0+n_1-1}\,1$ and once an entry in B becomes nonzero it is never changed, (ii) during phase h we only write to B the value h, (iii) because of the test at Line 4 the values written during phase h influence the algorithm only in subsequent phases. We maintain also an array $\mathsf{Block\_id}[1,\sigma]$ such that $\mathsf{Block\_id}[c]$ is the id of the block of $Z^{(h-1)}$ to which the last seen occurrence of symbol c belonged.

```
Initialize arrays $F[1,\sigma]$ and $\mathsf{Block\_id}[1,\sigma]$
$k_0 \leftarrow 1$; $k_1 \leftarrow 1$                          ▷ Init counters for $\mathsf{bwt}_0$ and $\mathsf{bwt}_1$
for $k \leftarrow 1$ to $n_0 + n_1$ do
    if $B[k] \neq 0$ and $B[k] \neq h$ then
        $\mathsf{id} \leftarrow k$                                ▷ A new block of $Z^{(h-1)}$ is starting
    end if
    $b \leftarrow Z^{(h-1)}[k]$                                   ▷ Read bit $b$ from $Z^{(h-1)}$
    if $b = 0$ then                  ▷ Get symbol from $\mathsf{bwt}_0$ or $\mathsf{bwt}_1$ according to $b$
        $c \leftarrow \mathsf{bwt}_0[k_0{+}{+}]$
    else
        $c \leftarrow \mathsf{bwt}_1[k_1{+}{+}]$
    end if
    $j \leftarrow F[c]{+}{+}$                ▷ Get destination for $b$ according to symbol $c$
    $Z^{(h)}[j] \leftarrow b$                                     ▷ Copy bit $b$ to $Z^{(h)}$
    if $\mathsf{Block\_id}[c] \neq \mathsf{id}$ then
        $\mathsf{Block\_id}[c] \leftarrow \mathsf{id}$                       ▷ Update block id for symbol $c$
        if $B[j] = 0$ then
            $B[j] = h$                                ▷ A new block of $Z^{(h)}$ will start here
        end if
    end if
end for
```

Fig. 3. Main loop of the H&M algorithm modified for the computation of the lcp values. At Line 1 for each symbol c we set $\mathsf{Block\_id}[c] = -1$ and $F[c]$ as in Fig. 2. At the beginning of the algorithm we initialize the array $B[0, n_0 + n_1]$ as $B = 1\,0^{n_0+n_1-1}\,1$.

The following lemma shows that the nonzero values of B at the end of phase h mark the boundaries of $Z^{(h)}$'s logical blocks.

Lemma 3. *For any $h \geq 0$, let ℓ, m be such that $1 \leq \ell \leq m \leq n_0 + n_1$ and*

$$\mathsf{lcp}_{01}[\ell] < h, \quad \min(\mathsf{lcp}_{01}[\ell+1], \ldots, \mathsf{lcp}_{01}[m]) \geq h, \quad \mathsf{lcp}_{01}[m+1] < h. \tag{4}$$

Then, at the end of phase h the array B is such that

$$B[\ell] \neq 0, \quad B[\ell+1] = \cdots = B[m] = 0, \quad B[m+1] \neq 0 \tag{5}$$

and $Z^{(h)}[\ell, m]$ is one of the blocks in (3). □

Proof. We prove the result by induction on h. For $h = 0$, hence before the execution of the first phase, (4) is only valid for $\ell = 1$ and $m = n_0 + n_1$ (recall we defined $\mathsf{lcp}_{01}[1] = \mathsf{lcp}_{01}[n_0 + n_1 + 1] = -1$). Since initially $B = 1\,0^{n_0+n_1-1}\,1$ our claim holds.

Suppose now that (4) holds for some $h > 0$. Let $s = \mathsf{t}_{01}[\mathsf{sa}_{01}[\ell], \mathsf{sa}_{01}[\ell]+h-1]$; by (4) s is a common prefix of the suffixes starting at positions $\mathsf{sa}_{01}[\ell]$, $\mathsf{sa}_{01}[\ell+1]$, ..., $\mathsf{sa}_{01}[m]$, and no other suffix of t_{01} is prefixed by s. By Property 1 the **0**s and **1**s in $Z^{(h)}[\ell, m]$ corresponds to the same set of suffixes That is, if $\ell \leq v \leq m$ and $Z^{(h)}[v]$ is the ith **0** (resp. jth **1**) of $Z^{(h)}$ then the suffix starting at $\mathsf{t}_0[\mathsf{sa}_0[i]]$ (resp. $\mathsf{t}_1[\mathsf{sa}_1[j]]$) is prefixed by s.

To prove (5) we start by showing that, if $\ell < m$, then at the end of phase $h-1$ it is $B[\ell+1] = \cdots = B[m] = 0$. To see this observe that the range $\mathsf{sa}_{01}[\ell, m]$ is part of a (possibly) larger range $\mathsf{sa}_{01}[\ell', m']$ containing all suffixes prefixed by the length $h-1$ prefix of s. By inductive hypothesis, at the end of phase $h-1$ it is $B[\ell'+1] = \cdots = B[m'] = 0$ which proves our claim since $\ell' \leq \ell$ and $m \leq m'$.

To complete the proof, we need to show that during phase h: (i) we do not write a nonzero value in $B[\ell+1, m]$ and (ii) we write a nonzero to $B[\ell]$ and $B[m+1]$ if they do not already contain a nonzero. Let $c = s[0]$ and $s' = s[1, h-1]$ so that $s = cs'$. Consider now the range $\mathsf{sa}_{01}[e, f]$ containing the suffixes prefixed by s'. By inductive hypothesis at the end of phase $h-1$ it is

$$B[e] \neq 0, \quad B[e+1] = \cdots = B[f] = 0, \quad B[f+1] \neq 0. \tag{6}$$

During iteration h, the bits in $Z^{(h)}[\ell, m]$ are possibly changed only when we are scanning the region $Z^{(h-1)}[e, f]$ and we find an entry $b = Z^{(h-1)}[k]$, $e \leq k \leq f$, such that the corresponding value in bwt_b is c. Note that by (6) as soon as k reaches e the variable id changes and becomes different from all values stored in $\mathsf{Block\_id}$. Hence, at the first occurrence of symbol c the value h will be stored in $B[\ell]$ (Line 18) unless a nonzero is already there. Again, because of (6), during the scanning of $Z^{(h-1)}[e, f]$ the variable id does not change so subsequent occurrences of c will not cause a nonzero value to be written to $B[\ell+1, m]$. Finally, as soon as we leave region $Z^{(h-1)}[e, f]$ and k reaches $f+1$, the variable id changes again and at the next occurrence of c a nonzero value will be stored in $B[m+1]$. If there are no more occurrences of c after we leave region $Z^{(h-1)}[e, f]$ then either $\mathsf{sa}_{01}[m+1]$ is the first suffix array entry prefixed by symbol $c+1$ or $m+1 = n_0 + n_1 + 1$. In the former case $B[m+1]$ gets a nonzero value at phase 1, in the latter case $B[m+1]$ gets a nonzero value when we initialize array B.

This completes the proof. □

Corollary 4. *For $i = 2, \ldots, n_0 + n_1$, if $\mathsf{lcp}_{01}[i] = \ell$, then starting from the end of phase $\ell + 1$ it is $B[i] = \ell + 1$.*

Proof. By Lemma 3 we know that $B[i]$ becomes nonzero only after phase $\ell+1$. Since at the end of phase ℓ it is still $B[i] = 0$ during phase $\ell + 1$ $B[i]$ gets the value $\ell + 1$ which is never changed in successive phases. □

The above corollary suggests the following algorithm to compute bwt_{01} and lcp_{01}: repeat the procedure of Fig. 3 until the phase h in which all entries in B become nonzero. At that point $Z^{(h)}$ describes how bwt_0 and bwt_1 should be merged to get bwt_{01} and for $i = 2, \ldots, n_0 + n_1$ $\mathsf{lcp}_{01}[i] = B[i] - 1$. The above strategy requires a number of iterations, each one taking $\mathcal{O}(n_0+n_1)$ time, equal to the maximum of the lcp values, for an overall complexity of $\mathcal{O}((n_0+n_1)\mathsf{maxlcp}_{01})$, where $\mathsf{maxlcp}_{01} = \max_i \mathsf{lcp}_{01}[i]$. In the next section we describe a much faster algorithm that avoids to re-process the portions of B and $Z^{(h)}$ which are no longer relevant for the computation of the final result.

4 The Gap Algorithm

Definition 5. *If $B[\ell] \neq 0$, $B[m+1] \neq 0$ and $B[\ell+1] = \cdots = B[m] = 0$, we say that block $Z^{(h)}[\ell, m]$ is* monochrome *if it contains only* **0***'s or only* **1***'s.* □

Since a monochrome block only contains suffixes from either t_0 or t_1, whose relative order and LCP's are known, it does not need to be further modified. This intuition is formalized by the following lemmas.

Lemma 6. *If at the end of phase h bit vector $Z^{(h)}$ contains only monochrome blocks we can compute bwt_{01} and lcp_{01} in $\mathcal{O}(n_0 + n_1)$ time.*

Proof. By Property 1, if we identify the i-th **0** in $Z^{(h)}$ with $\mathsf{bwt}_0[i]$ and the j-th **1** with $\mathsf{bwt}_1[j]$ the only elements which could be not correctly sorted by context are those within the same block. However, if the blocks are monochrome all elements belong to either bwt_0 or bwt_1 so their relative order is correct.

To compute lcp_{01} we observe that if $B[i] \neq 0$ then by (the proof of) Corollary 4 it is $\mathsf{lcp}_{01}[i] = B[i] - 1$. If instead $B[i] = 0$ we are inside a block hence $\mathsf{sa}_{01}[i-1]$ and $\mathsf{sa}_{01}[i]$ belong to the same string t_0 or t_1 and their LCP is directly available in lcp_0 or lcp_1. □

Lemma 7. *Suppose that, at the end of phase h, $Z^{(h)}[\ell, m]$ is a monochrome block. Then (i) for $g > h$, $Z^{(g)}[\ell, m] = Z^{(h)}[\ell, m]$, and (ii) processing $Z^{(h)}[\ell, m]$ during phase $h+1$ creates a set of monochrome blocks in $Z^{(h+1)}$.*

Proof. The first part of the Lemma follows from the observation that subsequent phases of the algorithm will only reorder the values within a block (and possibly create new sub-blocks); but if a block is monochrome the reordering will not change its actual content.

For the second part, we observe that during phase $h+1$ as k goes from ℓ to m the algorithm writes to $Z^{(h+1)}$ the same value which is in $Z^{(h)}[\ell, m]$. Hence, a new monochrome block will be created for each distinct symbol encountered (in bwt_0 or bwt_1) as k goes through the range $[\ell, m]$. □

The lemma implies that, if block $Z^{(h)}[\ell, m]$ is monochrome at the end of phase h, starting from phase $g = h+2$ processing the range $[\ell, m]$ will not change $Z^{(g)}$ with respect to $Z^{(g-1)}$. Indeed, by the lemma the monochrome blocks created in phase $h+1$ do not change in subsequent phases (in a subsequent phase a monochrome block can be split in sub-blocks, but the actual content of the bit vector does not change). The above observation suggests that, after we have processed block $Z^{(h+1)}[\ell, m]$ in phase $h+1$, we can mark it as *irrelevant* and avoid to process it again. As the computation goes on, more and more blocks become irrelevant. Hence, in the generic phase h instead of processing the whole $Z^{(h-1)}$ we process only the blocks which are still "active" and skip irrelevant blocks. Adjacent irrelevant blocks are merged so that among two active blocks there is at most one irrelevant block (the *gap* that gives the name to the algorithm). The overall structure of a single phase is shown in Fig. 4. The algorithm terminates

```
1: if (next block is irrelevant) then
2:     skip it
3: else
4:     process block
5:     if (processed block is monochrome) then
6:         mark it irrelevant
7:     end if
8: end if
9: if (last two blocks are irrelevant) then
10:    merge them
11: end if
```

Fig. 4. Main loop of the Gap algorithm. The processing of active blocks at Line 4 is done as in Lines 7–20 of Fig. 3.

when there are no more active blocks since this implies that all blocks have become monochrome and by Lemma 6 we are able to compute bwt_{01} and lcp_{01}.

We point out that at Line 2 of the Gap algorithm we cannot simply skip an irrelevant block ignoring its content. To keep the algorithm consistent we must correctly update the global variables of the main loop, i.e. the array F and the pointers k_0 and k_1 in Fig. 3. To this end a simple approach is to store for each irrelevant block the number of occurrences o_c of each symbol $c \in \Sigma$ in it and the pair (r_0, r_1) providing the number of $\mathbf{0}$'s and $\mathbf{1}$'s in the block (recall an irrelevant block may consist of adjacent monochrome blocks coming from different strings). When the algorithm reaches an irrelevant block, F, k_0, k_1 are updated setting $k_0 \leftarrow k_0 + r_0$, $k_1 \leftarrow k_1 + r_1$ and $\forall c\ F[c] \leftarrow F[c] + o_c$.

The above scheme for handling irrelevant blocks is simple and probably effective in most cases. However, using $\mathcal{O}(\sigma)$ time to skip an irrelevant block is not competitive for large alphabets. A better alternative is to build a wavelet tree for bwt_0 and bwt_1 at the beginning of the algorithm. Then, for each irrelevant block we store only the pair (r_0, r_1). When we reach an irrelevant block we use such pair to update k_0 and k_1. The array F is not immediately updated: Instead we maintain two global arrays $L_0[1,\sigma]$ and $L_1[1,\sigma]$ such that $L_0[c]$ and $L_1[c]$ store the value of k_0 and k_1 at the time the value $F[c]$ was last updated. At the *first* occurrence of a symbol c inside an active block we update $F[c]$ adding to it the number of occurrences of c in $\mathsf{bwt}_0[L_o[c]+1, k_0]$ and $\mathsf{bwt}_1[L_1[c]+1, k_1]$ that we compute in $\mathcal{O}(\log\sigma)$ time using the wavelet trees. Using this lazy update mechanism, handling irrelevant blocks adds a $\mathcal{O}(\min(\ell,\sigma)\log\sigma)$ additive slowdown to the cost of processing an active block of length ℓ.

Theorem 8. *Given* $\mathsf{bwt}_0, \mathsf{lcp}_0$ *and* $\mathsf{bwt}_1, \mathsf{lcp}_1$ *the* Gap *algorithm computes* bwt_{01} *and* lcp_{01} *in* $\mathcal{O}(\log(\sigma)(n_0 + n_1)\mathsf{avelcp}_{01})$ *time, where* $\mathsf{avelcp}_{01} = (\sum_i \mathsf{lcp}_{01}[i])/(n_0 + n_1)$ *is the average LCP of the string* t_{01}.

Proof. The correctness follows from the above discussion. For the analysis of the running time we reason as in [10] and observe that the sum, over all phases, of the length of all active blocks is bounded by $\mathcal{O}(\sum_i \mathsf{lcp}_{01}[i]) = \mathcal{O}((n_0 + n_1)\mathsf{avelcp}_{01})$. In any phase, using the lazy update mechanism, the cost of processing an active block of length ℓ is bounded by $\mathcal{O}(\ell \log(\sigma))$ and the time bound follows. $\square$

We point out that our Gap algorithm is related to the H&M algorithm as described in [10, Sect. 2.1]: Indeed, the sorting operations are essentially the same in the two algorithms. The main difference is that Gap keeps explicit track of the irrelevant blocks while H&M keeps explicit track of the active blocks (called buckets in [10]): this difference makes the non-sorting operations completely different. An advantage of working with irrelevant blocks is that they can be easily merged, while this is not the case for the active blocks in H&M. Of course, the main difference is that Gap merges simultaneously BWT *and* LCP values.

If we are simultaneously merging k BWTs, the only change in the algorithm is that the arrays $Z^{(h)}$ must now store integers in $[1, k]$; the overall running time is still $\mathcal{O}(n \log(\sigma)\mathsf{avelcp})$ where $n = \sum_i n_i$ is the size of the merged BWT and avelcp is the average of the values in the merged LCP array.

We now analyze the space usage of Gap when merging k BWTs. Let n denote the size of the merged BWTs. The arrays $\mathsf{bwt}_1, \ldots, \mathsf{bwt}_k$ take overall $n\lceil \log \sigma \rceil$ bits. At the end of the computation, in $\mathcal{O}(n)$ time using $Z^{(h)}$ the merged BWT can be written directly to disk or, using an in-place merging algorithm [8], overwritten to the space used by $\mathsf{bwt}_1, \ldots, \mathsf{bwt}_k$. The array B stores lcp values hence it can be represented in $n\lceil \log L \rceil$ bits, where $L = \max_i n_i$. Note that B takes the same space as the final merged LCP array, which indeed, at the end of the computation, could be overwritten to it using $Z^{(h)}$ (the merged LCP can also be written directly to the output file). In addition to the space used for BWT and LCP values, the algorithm uses $2n\lceil \log k \rceil$ bits for the arrays $Z^{(h)}$ (we only need 2 of them), and $\mathcal{O}(\sigma \log n)$ bits for the arrays F and $\mathsf{Block\_id}$.

The overall space usage so far is therefore $n(\lceil \log \sigma \rceil + \lceil \log L \rceil + 2\lceil \log k \rceil) + \mathcal{O}(\sigma \log n)$ bits. The only additional space used by the algorithm is the one used to keep track of the irrelevant blocks, which unfortunately cannot be estimated in advance since it depends on the maximum number of such blocks. In the worst case we can have $\Theta(n)$ blocks and the additional space can be $\Theta(nk \log n)$ bits. Although this is a rather unlikely possibility, it is important to have some form of control on this additional space. We use the following simple heuristic: we choose a threshold s and we keep track of an irrelevant block only if its size is at least s. This strategy introduces a $\mathcal{O}(s)$ time slowdown but ensures that there are at most $n/(s+1)$ irrelevant blocks simultaneously. In the next section we experimentally measure the influence of s on the space and running time of the algorithm and show that in practice the space used to keep track of irrelevant blocks is less than 10% of the total.

Note that also in [10] the authors faced the problem of limiting the memory used to keep track of the active blocks. They suggested the heuristic of keeping track of active blocks only after the h-th iteration ($h = 20$ for their dataset).

Table 1. Collections used in our experiments sorted by average LCP. Columns 4 and 5 refer to the lengths of the single documents. Pacbio are NGS reads from a *D.melanogaster* dataset. Illumina are NGS reads from Human ERA015743 dataset. Wiki-it are pages from Italian Wikipedia. Proteins are protein sequences from Uniprot. Collections and source files are available on https://people.unipmn.it/manzini/gap.

Name	Size GB	σ	Max Len	Ave Len	Max LCP	Ave LCP
Pacbio	6.24	5	40212	9567.43	1055	17.99
Illumina	7.60	6	103	102.00	102	27.53
Wiki-it	4.01	210	553975	4302.84	93537	61.02
Proteins	6.11	26	35991	410.22	25065	100.60

Table 2. For each collection we report the number k of subcollections, the average running time of gSACA-K+Φ in μsecs per symbol, and the running time (μsecs) and space usage (bytes) per symbol for Gap for different values of the s parameter.

Name	k	gSACA-K+Φ	$s = 50$		$s = 100$		$s = 200$	
			time	space	time	space	time	space
Pacbio	7	0.46	0.41	4.35	0.46	4.18	0.51	4.09
Illumina	4	0.48	0.93	3.31	1.02	3.16	1.09	3.08
Wiki-it	5	0.41	—	—	—	—	3.07	6.55
Proteins	4	0.59	3.90	4.55	5.18	4.29	7.05	4.15

5 Experimental Results

We have implemented the Gap algorithm in C and tested it on a desktop with 32 GB RAM and eight Intel-I7 3.40 GHz CPUs. All tests used a single CPU. We used the collections shown in Table 1. We represented LCP values using 1 byte for Illumina, 2 bytes for Pacbio and Proteins, and 4 bytes for Wiki-it. We always used 1 byte for each BWT value. We used n bytes to represent a pair of $Z^{(h)}$ arrays using 4 bits for each entry so that our implementation can merge simultaneously up to 16 BWTs. We used the simple strategy for skipping irrelevant blocks, i.e. we did not use wavelet trees to represent the input BWTs.

Referring to Table 2, we split each collection into k subcollections of size less than 2 GB and we computed the multi-string SA of each subcollection using gSACA-K [15]. From the SA we computed the multi-string BWT and LCP arrays using the Φ algorithm [12] (implemented in gSACA-K). This computation used 13 bytes per input symbol. Then, we merged the subcollections multi-string BWTs and LCPs using Gap with different values of the parameter s which determines the size of the smallest irrelevant block we keep track of. Note that for Wiki-it s has no influence since the algorithm never keeps track of a block smaller than $\sigma+k$. The rationale is that in our implementation skipping a block takes $\mathcal{O}(\sigma+k)$ time, so there is no advantage in skipping a block smaller than that size.

From the results in Table 2 we see that Gap running time is indeed roughly proportional to the average LCP. For example, Pacbio and Illumina collections both consist of DNA reads but, despite Pacbio reads being longer and having a larger maximum LCP, Gap is twice as fast on them because of the smaller average LCP. Similarly, Gap is faster on Wiki-it than on Proteins despite the latter collection having a smaller alphabet and shorter documents. gSACA-K running time is not significantly influenced by the average LCP. If we compare Gap with gSACA-K we see that only in one instance, Pacbio with $s = 50$, Gap is faster than gSACA-K in terms of μsecs per input symbol. However, since Gap is designed to post-process gSACA-K output, the comparison of the running time is only important to the extent Gap is not a bottleneck in our two-step strategy to compute the multi-string BWT and LCP arrays: the experiments show this is not the case. We point out that on our 32 GB machine, gSACA-K cannot compute the multi-string SA for any of the collections since for inputs larger that 2 GB it uses 9 bytes per input symbol.

As expected, the parameter s offers a time-space tradeoff for the Gap algorithm. In the space reported in Table 2, the fractional part is the peak space usage for irrelevant blocks, while the whole value is the space used by the arrays bwt_i, B and $Z^{(h)}$. For example, for Wiki-it we use n bytes for the BWTs, $4n$ bytes for the LCP values (the B array), n bytes for $Z^{(h)}$, and the remaining $0.55n$ bytes are mainly used for keeping track of irrelevant blocks. This is a relatively high value since in our current implementation the storage of a block grows linearly with the alphabet size. For DNA sequences and $s = 200$ the cost of storing blocks is less than 3% of the total without a significant slowdown in the running time.

For completeness, we tested the H&M implementation from [10] on the Pacbio collection. The running time was 14.57 μsecs per symbol and the space usage 2.28 bytes per symbol. These values are only partially significative for several reasons: (i) H&M computes the BWT from scratch, hence doing also the work of gSACA-K, (ii) H&M doesn't compute the LCP array, hence the lower space usage, (iii) the algorithm is implemented in Cython which makes it easier to use in a Python environment but is not as fast and space efficient as C.

Appendix

Proof of Lemma 2: We prove the result by induction. For $h = 0$, $\delta = 0, 1$ $\mathsf{t}_\delta[\mathsf{sa}_\delta[i], \mathsf{sa}_\delta[i] - 1]$ is the empty string so (2) is always true and Property 1 is satisfied by $Z^{(0)} = \mathbf{0}^{n_0}\mathbf{1}^{n_1}$.

To prove the "if" part, let $h > 0$ and let $1 \leq v < w \leq n_0 + n_1$ denote two indexes such that $Z^{(h)}[v]$ is the i-th $\mathbf{0}$ and $Z^{(h)}[w]$ is the j-th $\mathbf{1}$ in $Z^{(h)}$. We need to show that under these assumptions inequality (2) on the lexicographic order holds.

Assume first $\mathsf{t}_0[\mathsf{sa}_0[i]] \neq \mathsf{t}_1[\mathsf{sa}_1[j]]$. The hypothesis $v < w$ implies $\mathsf{t}_0[\mathsf{sa}_0[i]] < \mathsf{t}_1[\mathsf{sa}_1[j]]$ hence (2) certainly holds.

Assume now $\mathsf{t}_0[\mathsf{sa}_0[i]] = \mathsf{t}_1[\mathsf{sa}_1[j]]$. We preliminarily observe that it must be $\mathsf{sa}_0[i] \neq n_0$ and $\mathsf{sa}_1[i] \neq n_1$: otherwise we would have $\mathsf{t}_0[\mathsf{sa}_0[i]] = \$_0$ or $\mathsf{t}_1[\mathsf{sa}_1[j]] = \$_1$ which is impossible since these symbols appear only once in t_0 and t_1.

Let v', w' denote respectively the value of the main loop variable k in the procedure of Fig. 2 when the entries $Z^{(h)}[v]$ and $Z^{(h)}[w]$ are written (hence, during the scanning of $Z^{(h-1)}$). The hypothesis $v < w$ implies $v' < w'$. By construction $Z^{(h-1)}[v'] = \mathbf{0}$ and $Z^{(h-1)}[w'] = \mathbf{1}$. Say v' is the i'-th $\mathbf{0}$ in $Z^{(h-1)}$ and w' is the j'-th $\mathbf{1}$ in $Z^{(h-1)}$. By the inductive hypothesis on $Z^{(h-1)}$ we have

$$\mathsf{t}_0[\mathsf{sa}_0[i'], \mathsf{sa}_0[i'] + h - 2] \preceq \mathsf{t}_1[\mathsf{sa}_1[j'], \mathsf{sa}_1[j'] + h - 2], \tag{7}$$

The fundamental observation is that, being $\mathsf{sa}_0[i] \neq n_0$ and $\mathsf{sa}_1[i] \neq n_1$, it is

$$\mathsf{sa}_0[i'] = \mathsf{sa}_0[i] + 1 \qquad \text{and} \qquad \mathsf{sa}_1[j'] = \mathsf{sa}_1[j] + 1.$$

Since

$$\mathsf{t}_0[\mathsf{sa}_0[i], \mathsf{sa}_0[i] + h - 1] = \mathsf{t}_0[\mathsf{sa}_0[i]]\mathsf{t}_0[\mathsf{sa}_0[i'], \mathsf{sa}_0[i'] + h - 2] \tag{8}$$

$$\mathsf{t}_1[\mathsf{sa}_1[j], \mathsf{sa}_1[j] + h - 1] = \mathsf{t}_1[\mathsf{sa}_1[j]]\mathsf{t}_1[\mathsf{sa}_1[j'], \mathsf{sa}_1[j'] + h - 2] \tag{9}$$

combining $\mathsf{t}_0[\mathsf{sa}_0[i]] = \mathsf{t}_1[\mathsf{sa}_1[j]]$ with (7) gives us (2).

For the "only if" part assume (2) holds. We need to prove that in $Z^{(h)}$ the i-th $\mathbf{0}$ precedes the j-th $\mathbf{1}$. If $\mathsf{t}_0[\mathsf{sa}_0[i]] < \mathsf{t}_1[\mathsf{sa}_1[j]]$ the proof is immediate. If $\mathsf{t}_0[\mathsf{sa}_0[i]] = \mathsf{t}_1[\mathsf{sa}_1[j]]$, we must have

$$\mathsf{t}_0[\mathsf{sa}_0[i] + 1, \mathsf{sa}_0[i] + h - 1] \preceq \mathsf{t}_1[\mathsf{sa}_1[j] + 1, \mathsf{sa}_1[j] + h - 1].$$

By induction, if $\mathsf{sa}_0[i'] = \mathsf{sa}_0[i] + 1$ and $\mathsf{sa}_1[j'] = \mathsf{sa}_1[j] + 1$ in $Z^{(h-1)}$ the i'-th $\mathbf{0}$ precedes the j'-th $\mathbf{1}$. During phase h, the i-th $\mathbf{0}$ in $Z^{(h)}$ is written when processing the i'-th $\mathbf{0}$ of $Z^{(h-1)}$, and the j-th $\mathbf{1}$ in $Z^{(h)}$ is written when processing the j'-th $\mathbf{1}$ of $Z^{(h-1)}$. Since in $Z^{(h-1)}$ the i'-th $\mathbf{0}$ precedes the j'-th $\mathbf{1}$ and

$$\mathsf{bwt}_0[i'] = \mathsf{t}_0[\mathsf{sa}_0[i]] = \mathsf{t}_1[\mathsf{sa}_1[j]] = \mathsf{bwt}_1[j']$$

in $Z^{(h)}$ their relative order does not change and the i-th $\mathbf{0}$ precedes the j-th $\mathbf{1}$ as claimed. □

References

1. Belazzougui, D.: Linear time construction of compressed text indices in compact space. In: STOC, pp. 148–193. ACM (2014)
2. Bonizzoni, P., Vedova, G.D., Nicosia, S., Previtali, M., Rizzi, R.: A new lightweight algorithm to compute the BWT and the LCP array of a set of strings. CoRR abs/1607.08342 (2016)
3. Bonizzoni, P., Vedova, G.D., Pirola, Y., Previtali, M., Rizzi, R.: Computing the BWT and LCP array of a set of strings in external memory. CoRR abs/1705.07756 (2017)
4. Burkhardt, S., Kärkkäinen, J.: Fast lightweight suffix array construction and checking. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 55–69. Springer, Heidelberg (2003). doi:10.1007/3-540-44888-8_5

5. Cox, A.J., Garofalo, F., Rosone, G., Sciortino, M.: Lightweight LCP construction for very large collections of strings. J. Discrete Algorithms **37**, 17–33 (2016)
6. Ferragina, P., Gagie, T., Manzini, G.: Lightweight data indexing and compression in external memory. In: López-Ortiz, A. (ed.) LATIN 2010. LNCS, vol. 6034, pp. 697–710. Springer, Heidelberg (2010). doi:10.1007/978-3-642-12200-2_60
7. Ferragina, P., Gagie, T., Manzini, G.: Lightweight data indexing and compression in external memory. Algorithmica (2011)
8. Geffert, V., Gajdos, J.: Multiway in-place merging. Theor. Comput. Sci. **411**(16–18), 1793–1808 (2010)
9. Gog, S., Ohlebusch, E.: Compressed suffix trees: efficient computation and storage of LCP-values. ACM J. Exp. Algorithmics **18** (2013). http://doi.acm.org/10.1145/2444016.2461327
10. Holt, J., McMillan, L.: Constructing Burrows-Wheeler transforms of large string collections via merging. In: BCB, pp. 464–471. ACM (2014)
11. Holt, J., McMillan, L.: Merging of multi-string BWTs with applications. Bioinformatics **30**(24), 3524–3531 (2014)
12. Kärkkäinen, J., Manzini, G., Puglisi, S.J.: Permuted longest-common-prefix array. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009. LNCS, vol. 5577, pp. 181–192. Springer, Heidelberg (2009). doi:10.1007/978-3-642-02441-2_17
13. Kärkkäinen, J., Kempa, D.: LCP array construction in external memory. ACM J. Exp. Algorithmics **21**(1), 1.7:1–1.7:22 (2016)
14. Léonard, M., Mouchard, L., Salson, M.: On the number of elements to reorder when updating a suffix array. J. Discrete Algorithms **11**, 87–99 (2012). http://dx.doi.org/10.1016/j.jda.2011.01.002
15. Louza, F.A., Gog, S., Telles, G.P.: Induced suffix sorting for string collections. In: DCC, pp. 43–52. IEEE (2016)
16. Louza, F.A., Gog, S., Telles, G.P.: Inducing enhanced suffix arrays for string collections. Theor. Comput. Sci. **678**, 22–39 (2017)
17. Louza, F.A., Telles, G.P., Ciferri, C.D.A.: External memory generalized suffix and LCP arrays construction. In: Fischer, J., Sanders, P. (eds.) CPM 2013. LNCS, vol. 7922, pp. 201–210. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38905-4_20
18. Mantaci, S., Restivo, A., Rosone, G., Sciortino, M.: An extension of the Burrows-Wheeler transform. Theor. Comput. Sci. **387**(3), 298–312 (2007)
19. Manzini, G.: Two space saving tricks for linear time LCP computation. In: Proceedings of 9th Scandinavian Workshop on Algorithm Theory (SWAT 2004), pp. 372–383. Springer-Verlag, LNCS n. 3111 (2004)
20. Manzini, G., Ferragina, P.: Engineering a lightweight suffix array construction algorithm. In: Möhring, R., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 698–710. Springer, Heidelberg (2002). doi:10.1007/3-540-45749-6_61
21. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comput. Surv. **39**(1), Article no. 2 (2007). doi:10.1145/1216370.1216372
22. Nong, G.: Practical linear-time O(1)-workspace suffix sorting for constant alphabets. ACM Trans. Inf. Syst. **31**(3), Article no. 15 (2013). doi:10.1145/2493175.2493180
23. Sirén, J.: Compressed suffix arrays for massive data. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 63–74. Springer, Heidelberg (2009). doi:10.1007/978-3-642-03784-9_7
24. Sirén, J.: Burrows-wheeler transform for Terabases. In: IEEE Data Compression Conference (DCC), pp. 211–220 (2016)