Computing the BWT and LCP array of a Set of Strings in External Memory

Paola Bonizzoni  Gianluca Della Vedova  Yuri Pirola
Marco Previtali  Raffaella Rizzi
DISCo, Univ. Milano-Bicocca, Milan, Italy

Abstract

Indexing very large collections of strings, such as those produced by the widespread next generation sequencing technologies, heavily relies on multi-string generalization of the Burrows-Wheeler Transform (BWT): recent developments in this field have resulted in external memory algorithms, motivated by the large requirements of in-memory approaches.

The related problem of computing the Longest Common Prefix (LCP) array of a set of strings is often instrumental in several algorithms: for example, to compute the suffix-prefix overlaps among strings, which is an essential step for many genome assembly algorithms.

In this paper we propose a new external memory method to simultaneously build the BWT and the LCP array on a collection of \( m \) strings of length \( k \) with \( \mathcal{O}(mkl) \) time and I/O complexity, using \( \mathcal{O}(k + m) \) main memory, where \( l \) is the maximum value in the LCP array.

1 Introduction

In this paper we address the problem of constructing in external memory the Burrows-Wheeler Transform (BWT) and the Longest Common Prefix (LCP) array for a large collection of strings. The widespread use of Next-Generation Sequencing (NGS) technologies, that are producing everyday several terabytes of data that has to be analyzed, requires efficient strategies to index very large collections of strings. For example, common applications in metagenomics require indexing of collections of strings (reads) that are sampled from several genomes: those strings can easily contain more than \( 10^8 \) characters. In fact, to start a catalogue of the human gut microbiome, more than 500GB of data have been used [22].

The Burrows-Wheeler Transform (BWT) [7] is a reversible transformation of a text that was originally designed for text compression; it is used for example in the bzip2 program. The BWT of a text \( T \) is a permutation of its symbols and is strictly related
to the Suffix Array of \( T \). In fact, the \( i \)-th symbol of the BWT is the symbol preceding the \( i \)-th smallest suffix of \( T \) according to the lexicographical sorting of the suffixes of \( T \). The Burrows-Wheeler Transform has gained importance beyond its initial purpose, and has become the basis for self-indexing structures such as the FM-index \[10\], which allows to efficiently perform important tasks such as searching a pattern in a text \[10, 14, 23\]. The generalization of the BWT (and the FM-index) to a collection of strings has been introduced in \[18\] and \[19\].

An entire generation of recent bioinformatics tools heavily rely on the notion of BWT. Representing the reference genome with its FM-index is the basis of the most widely used aligners, such as Bowtie \[12\], BWA \[15, 16\] and SOAP2 \[17\].

Still, to attack some other fundamental bioinformatics problems, such as genome assembly, an all-against-all comparison among the input strings is needed, especially to find all prefix-suffix matches (or overlaps) between reads in the context of the Overlap-Layout-Consensus (OLC) approach based on string graph \[20\]. This fact justifies to search for extremely efficient algorithms to compute the BWT on a collection of strings \[2, 9, 13, 27\]. For example, SGA (String Graph Assembler) \[25\] is a de novo genome assembler that builds a string graph from the FM-index of the collection of input reads. In a preliminary version of SGA \[24\], the authors estimated, for human sequencing data at a 20x coverage, the need of 700Gbytes of RAM in order to build the suffix array, using the construction algorithm in \[21\], and the FM-index. Another technical device that is used to tackle the genome assembly in the OLC approach is the Longest Common Prefix (LCP) array of a collection of strings, which is instrumental to compute (among others) the prefix-suffix matches in the collection.

The construction of the BWT (and LCP array) of a huge collection of strings is a challenging task. A simple approach is constructing the BWT from the suffix array, but it is prohibitive for massive datasets. A first attempt to solve this problem \[26\] partitions the input collection into batches, computes the BWT for each batch and then merges the results.

The huge amount of available biological data has stimulated the development of the first efficient external memory algorithms (called, BCR and BCRext) to construct the BWT of a collection of strings \[1\]. Similarly, a lightweight approach to the construction of the LCP array (called extLCP) has been investigated in \[8\]. With the ultimate goal of obtaining an external memory genome assembler, LSG \[4, 6\] is based on BCRext and contains an external memory approach to compute the string graph of a set of strings. In that approach, external memory algorithms to compute the BWT and the LCP array \[2, 3\] are fundamental.

In this paper we present a new lightweight (external memory) approach to compute the BWT and the LCP array of a collection of strings, which is alternative to extLCP \[8\] and other approaches \[5, 11\]. The algorithm BCRext is proposed together with BCR and both are designed to work on huge collections of strings (the experimental analysis is on billions of 100-long strings). Especially extLCP is lightweight because, on a collection of \( m \) strings of length \( k \), it uses only \( \mathcal{O}(m + \sigma^2) \) RAM space and essentially \( \mathcal{O}(mk^2) \) CPU time, with an I/O complexity that is \( \mathcal{O}(mk^2) \), under the usual assumption that the word
size is sufficiently large to store all addresses.

An important question is to achieve the optimal $O(km)$ I/O complexity. Both BCR and BCRext build the BWT with a column-wise approach, where at each step $i$ the elements preceding the suffixes of length $k - i - 1$ of each read are inserted into the correct positions of the partial BWT that considers only suffixes shorter than $k - i - 1$. Moreover, both algorithms are described as a succession of sequential scans, where the partial BWTs are read from and and written to external files, thus obtaining a small main memory footprint.

Our algorithm has an $O(mk)$ time complexity, and uses $O(mkl)$ I/O volume and $O(k + m)$ main memory, where $l$ is the maximum value in the LCP array. Moreover, our approach is entirely based on linear scans (i.e., it does not contain a sorting step) which makes it more amenable to actual disk-based implementations. We point out that $l \leq k$, therefore our time and I/O complexities are at least as good as those of extLCP [8]. The RAM usages of our approach and of extLCP are not comparable, since they are respectively $O(m + k)$ and $O(m + \sigma^2)$.

## 2 Preliminaries

Let $\Sigma = \{c_0, c_1, \ldots, c_\sigma\}$ be a finite alphabet where $c_0 = \$ \text{(called sentinel)}$, and $c_0 < c_1 < \cdots < c_\sigma$ where $<$ specifies the lexicographic ordering over alphabet $\Sigma$. We consider a collection $S = \{s_1, s_2, \ldots, s_m\}$ of $m$ strings, where each string $s_j$ consists of $k$ symbols over the alphabet $\Sigma \setminus \{$\$\}$ and is terminated by the symbol $\$\$.

The $i$-th symbol of string $s_j$ is denoted by $s_j[i]$, and the substring $s_j[i]s_j[i + 1] \cdots s_j[t]$ of $s_j$ is denoted by $s_j[i : t]$. We are assuming that all the strings in $S$ have the same length $k$ only to simplify the presentation: it is immediate to extend our algorithm to a generic set $S$ of strings.

Given a vector $V$, we denote with $V[1 : q]$ the first $q$ elements of $V$ and with $\text{rank}_V(q, x)$ the number of elements equals to $x$ in $V[1 : q]$.

The suffix and prefix of $s_j$ of length $l$ are the substrings $s_j[k - l + 1 : k]$ (denoted by $s_j[k - l + 1 : :])$ and $s_j[1 : l]$ (denoted by $s_j[:: l]$) respectively. Then the $l$-suffix and $l$-prefix of a string $s_j$ is the suffix and prefix with length $l$, respectively.

Given the lexicographic ordering $X$ of the suffixes of $S$, the Suffix Array is the $(m(k + 1))$-long array $SA$ where the element $SA[i]$ is equal to $(p, j)$ if and only if the $i$-th element of $X$ is the $p$-suffix of string $s_j$. The Burrows-Wheeler Transform (BWT) of $S$ is the $(m(k + 1))$-long array $B$ where if $SA[i] = (p, j)$, then $B[i]$ is the first symbol of the $(p + 1)$-suffix of $s_j$ if $p < k$, otherwise $B[i] = \$$$. In other words $B$ consists of the symbols preceding the ordered suffixes of $X$, where the symbol $\$$ is also the one that precedes each string $s_i \in S$. The Longest Common Prefix (LCP) array of $S$ is the $(m(k + 1))$-long array $LCP$ such that $LCP[i]$ is the length of the longest prefix shared by suffixes $X[i - 1]$ and $X[i]$. Conventionally, $LCP[1] = -1$.

Now, we give the definition of interleave of a generic set of arrays, that will be used extensively in the following.
Definition 1. Given \( n + 1 \) arrays \( V_0, V_1, \ldots, V_n \), then an array \( W \) is an interleave of \( V_0, V_1, \ldots, V_n \) if \( W \) is the result of merging the arrays such that: (i) there is a 1-to-1 function \( \psi_W \) from the set \( \bigcup_{i=0}^{n}\{(i, j) : 1 \leq j \leq |V_i|\} \) to the set \( \{q : 1 \leq q \leq |W| = \sum_{i} |V_i|\} \), (ii) \( V_i[j] = W[\psi_W(i, j)] \) for each \( i, j \), and (iii) \( \psi_W(i, j_1) < \psi_W(i, j_2) \) for each \( j_1 < j_2 \).

The interleave \( W \) is an array giving a fusion of \( V_0, V_1, \ldots, V_n \) which preserves the relative order of the elements in each one of the arrays. As a consequence, for each \( i \) with \( 0 \leq i \leq n \), the \( j \)-th element of \( V_i \) corresponds to the \( j \)-th occurrence in \( W \) of an element of \( V_i \). This fact allows to encode the function \( \psi_W \) as an array \( I_W \) such that \( I_W[q] = i \) if and only if \( W[q] \) is an element of \( V_i \). By observing that \( W[q] \) is equal to \( V_{I_W[q]}[j] \) where \( j = \text{rank}_{I_W}(q, I_W[q]) \), it is easy to show how to reconstruct \( W \) from \( I_W \) (see Algorithm 1 where the array \( \text{pos}[i] \) at line 6 is equal to \( \text{rank}_{I_W}(q, i) \)).

In the following, we will refer to vector \( I_W \) as interleave-encoding (or simply encoding). Figure 1 shows an example of an interleave of four arrays \( V_0, V_1, V_2, V_3 \) and its encoding.

![Figure 1: Example of an interleave \( W \) of four arrays \( V_0, V_1, V_2, V_3 \).](image)

|       |      |
|-------|------|
| **W** | **I_W** |
| T     | 0    |
| A     | 2    |
| A     | 3    |
| A     | 3    |
| C     | 1    |
| C     | 2    |
| C     | 2    |
| G     | 1    |
| G     | 1    |
| T     | 0    |
| A     | 0    |
| T     | 3    |

**Algorithm 1:** Reconstruct the interleave \( W \) from the encoding \( I_W \)

1. for \( i \leftarrow 0 \) to \( n \) do
2. \hspace{1em} \text{pos}[i] \leftarrow 0;
3. for \( q \leftarrow 1 \) to \( |I_W| \) do
4. \hspace{1em} \text{i} \leftarrow I_W[q];
5. \hspace{1em} \text{pos}[i] \leftarrow \text{pos}[i] + 1;
6. \hspace{1em} W[q] \leftarrow V_i[\text{pos}[i]];

![Algorithm 1: Reconstruct the interleave \( W \) from the encoding \( I_W \)](image)
3 The algorithm

In this section we will provide a sketch of our algorithm, Let $X_t$ and $B_t$ ($0 \leq t \leq k$) be arrays of length $m$ such that $X_t[i]$ is the $i$-th smallest $l$-suffix of $S$ and $B_t[i]$ is the symbol preceding $X_t[i]$. In particular $X_0$ and $B_0$ list (respectively) the 0-suffixes and the last characters of the input strings in their order in the set $S$. Observe that $B_t$ is a subsequence of the BWT $B$ of $S$, and it is easy to see that $B$ is an interleave of the $k+1$ arrays $B_0, B_1, \ldots, B_k$, since the ordering of symbols in $B_t$ ($0 \leq t \leq k$) is preserved in $B$.

Similarly, the lexicographic ordering $X$ of all suffixes of $S$ is an interleave of the arrays $X_0, X_1, \ldots, X_k$. Let $I_B$ be the encoding of the interleave of arrays $B_0, B_1, \ldots, B_k$ giving the BWT $B$, and let $I_X$ be the encoding of the interleave of arrays $X_0, X_1, \ldots, X_k$ giving $X$. Then it is possible to show that $I_B = I_X$. Now, given $I_B$ it is immediate to show that $B$ can be reconstructed as follows: for each $i$, where $1 \leq i \leq (k+1)m$, if $I_B[i] = l$ and $\text{rank}_{I_B}(i, l) = j$ then $B[i]$ is the character that precedes the $j$-th suffix in $X_i$. Indeed, the $i$-th suffix in the lexicographic ordering of $X$ is the $j$-th in $X_i$ of a string $s_t$ and thus $B[i] = s_t[k - l]$, when $0 \leq l \leq k - 1$, otherwise (when $l = k$) $B[i] = \$", and by definition of arrays $B_0, B_1, \ldots, B_k$, $B[i]$ is the $j$-th symbol of array $B_i$. In the following, we will denote $B_0, B_1, \ldots, B_k$ and $X_0, X_1, \ldots, X_k$ as partial BWTs and partial Suffix Arrays, respectively.

Figure 2 shows an example of partial BWTs and partial Suffix Arrays for a set of $m = 3$ reads, on alphabet $\{A, C, G, T\}$, having length $k = 4$, whose interleaves $B$ and $X$ (BWT and sorted suffixes, respectively), and the encoding $I_B = I_X$ are reported in the the first, second and third columns of Figure 3.

Our algorithm for building the BWT $B$ and the LCP array, differently from BCRext and extLCP \[1\] \[8\], consists of two distinct phases: in the first phase the partial BWTs $B_0, B_1, \ldots, B_k$ are computed (see Section 4), while the second phase (see Section 5) is based on the approach of Holt and McMillan \[11\] and determines $I_X$ (which is equal to $I_B$), thus allowing to reconstruct $B$ as an interleave of $B_0, B_1, \ldots, B_k$. Holt and McMillan propose an algorithmic approach to merge two BWTs. We apply that approach to merge the arrays $X_0, X_1, \ldots, X_k$ in order to compute the BWT and the LCP array of a set of strings. This method consists in applying a radix sort approach and is realized in external memory by using lists which can be implemented in files. Each list/file is sequentially accessed. Observe that our algorithm implicitly merges the arrays $X_0, X_1, \ldots, X_k$ and it does not need to compute those arrays explicitly.

4 Computing the partial BWTs

The input strings $s_1, \ldots, s_m$ are preprocessed in order to compute $k+1$ arrays $T_0, T_1, \ldots, T_k$ with length $m$, where $T_i$ lists the characters in position $k - l$ of the input strings, such that $T_i[l] = s_i[k - l]$ when $0 \leq l \leq k - 1$ and $T_i[l] = \$" when $l = k$. Observe that $T_i[l]$ is the symbol preceding the $l$-suffix of $s_i$. Clearly, those arrays can be computed in $O(km)$ time and $O(km \log \sigma)$ I/O complexity by reading sequentially the input strings.

The partial BWTs $B_0, B_1, \ldots, B_k$ are computed by Algorithm 2 by receiving in input
the arrays $T_0, T_1, \ldots, T_{k-1}$ and by using $k + 1$ arrays $N_l$ with length $m$ ($0 \leq l \leq k$), where $N_l[i] = q$ if and only if the $i$-th element of $X_l$ is the $l$-suffix of the input string $s_q$. The symbol $B_l[i]$, for $0 \leq l \leq k - 1$ precedes the $l$-suffix $s_q[k - l + 1:]$, that is $B_l[i] = s_q[k - l]$. When $l = k$, $B_k[i] = \$$. In particular, $N_0$ is the sequence of indexes $\langle 1, 2, 3, \ldots, |S| \rangle$ and $B_0$ is the sequence $\langle s_1[k], s_2[k], \ldots, s_m[k] \rangle$ of the last symbols of the input strings (i.e. the symbols before the sentinels), that is $B_0 = T_0$.

In order to specify the structure of Algorithm 2, given a symbol $c_h$ of the alphabet $\Sigma$, we define the $c_h$-projection operation $\Pi_{c_h}$ over the array $N_l$, with $l > 0$, that consists in taking from $N_l$ only the entries $q$ such that $s_q[k - l] = c_h$. In other words $\Pi_{c_h}(N_l)$ is the vector that projects the entries of $N_l$ corresponding to strings whose $l$-suffix is preceded by the symbol $c_h$. Then the following Lemma directly follows from definition of $N_{l-1}$.

**Lemma 1.** The sequence of indexes of strings whose $l$-suffix starts with symbol $c_h$ and ordered w.r.t. the $l$-suffix, is equal to vector $\Pi_{c_h}(N_{l-1})$.

The main consequence of the above Lemma the array $N_l$ can be simply obtained from $N_{l-1}$ as the concatenation $\Pi_{c_0}(N_{l-1}) \cdot \Pi_{c_1}(N_{l-1}) \cdots \Pi_{c_\sigma}(N_{l-1})$ where $c_0 \cdot c_1 \cdots c_\sigma$ is the lexicographic order of symbols of alphabet $\Sigma$. The $c_h$-projection of $N_{l-1}$, is computed by listing the entries $q$ at the positions $i$ of $N_{l-1}$ such that $B_{l-1}[i] = c_h$. Indeed, $B_{l-1}$ lists the symbols preceedings the ordered $(l - 1)$-suffixes.

Algorithm 2 computes the partial BWTs $B_0, \ldots, B_k$ in $k$ iterations. At iteration $l$, the arrays $B_l$ and $N_l$ are computed from arrays $B_{l-1}$ and $N_{l-1}$ (Figure 4 shows the iteration $l = 1$ for the set of reads of Figure 2). The $c_h$-projection of $N_{l-1}$ is stored (at each iteration) in the list $P(c_h)$ that is empty at the beginning of the iteration. Observe that each list is sequentially read and can be implemented in a file. At iteration $l$, the procedure first computes $N_l$ from $B_{l-1}$ and $N_{l-1}$ as follows. The arrays $B_{l-1}$ and $N_{l-1}$ are sequentially read and, for each position $i$, the value $q = N_{l-1}[i]$ is appended to the list $P(c_h)$, where $c_h$ is the symbol in position $i$ of $B_{l-1}$. Finally, the array $N_l$ is obtained as the concatenation of lists $P(c_0)P(c_1)\cdots P(c_\sigma)$. After computing $N_l$, the array $B_l$ can be obtained. Indeed, assuming that the $j$-th element in the ordered list of $l$-suffixes is the suffix of string $s_q$ (that is, $N_l[j] = q$) the symbol preceding such suffix is $s_q[k - l]$ and is directly obtained by accessing position $q$ of array $T_l$ (recall that $T_l[q]$ is the character $s_q[k - l]$ or the sentinel $\$ when $l = k$). The algorithm sequentially reads $N_l$ and, for each read value $q$, it sets $B_l[i]$ to the value $T_l[q]$. Observe that also the array $B_l$ and $N_l$ of each iteration can be stored in lists (implemented in files) since they are sequentially accessed. Due to a random access, each array $T_l$ is kept in RAM with a total space cost of $O(km \log \sigma)$.

## 5 Computing the encoding $I_X$ and the LCP array

This section is devoted to describe the second step of our algorithm which computes the BWT $B$ and the LCP array. The encoding $I_X$ of the interleave $X$ of the arrays
| $s_1$ | T C G T |
|-------|---------|
| $s_2$ | A C C T |
| $s_3$ | A A C A |

\[
\begin{array}{c|c|c}
& B_0 & X_0 \\
T & $ & \\
T & $ & \\
A & $ & \\
\end{array}
\begin{array}{c|c|c}
& B_1 & X_1 \\
C & A$ & \\
G & T$ & \\
C & T$ & \\
\end{array}
\begin{array}{c|c|c}
& B_2 & X_2 \\
A & CA$ & \\
C & CT$ & \\
C & GT$ & \\
\end{array}
\begin{array}{c|c|c}
& B_3 & X_3 \\
A & ACA$ & \\
A & CCT$ & \\
T & CGT$ & \\
\end{array}
\begin{array}{c|c|c}
& B_4 & X_4 \\
$ & AACA$ & \\
$ & ACCT$ & \\
$ & TCGT$ & \\
\end{array}
\]

Figure 2: An example of $m = 3$ reads $s_1, s_2, s_3$ of length $k = 4$, together with the partial BWTs $B_0, B_1, B_2, B_3, B_4$ and the partial Suffix Arrays $X_0, X_1, X_2, X_3, X_4$.

\[
\begin{array}{c|c|c|c}
B & X & I_B = I_X & LCP \\
T & $ & 0 & -1 & B_0[1] \ X_0[1] \\
T & $ & 0 & 0 & B_0[2] \ X_0[2] \\
A & $ & 0 & 0 & B_0[3] \ X_0[3] \\
C & A$ & 1 & 0 & B_1[1] \ X_1[1] \\
$ & AACA$ & 4 & 1 & B_4[1] \ X_4[1] \\
A & ACA$ & 3 & 1 & B_3[1] \ X_3[1] \\
$ & ACCT$ & 4 & 2 & B_4[2] \ X_4[2] \\
A & CA$ & 2 & 0 & B_2[1] \ X_2[1] \\
A & CCT$ & 3 & 1 & B_3[2] \ X_3[2] \\
T & CGT$ & 3 & 1 & B_3[2] \ X_3[2] \\
C & CT$ & 2 & 1 & B_2[2] \ X_2[2] \\
C & GT$ & 2 & 0 & B_3[3] \ X_3[3] \\
G & T$ & 1 & 0 & B_1[2] \ X_1[2] \\
C & T$ & 1 & 1 & B_1[3] \ X_1[3] \\
$ & TCGT$ & 4 & 1 & B_4[3] \ X_4[3] \\
\end{array}
\]

Figure 3: BWT $B$, sorted suffixes $X$, encoding $I_B = I_X$ and LCP array for the set of reads presented in Figure 2. The last two columns report, for each element of $B$ and $X$, its origin in arrays $B_l$ and $X_l$ (respectively).
(a) Input arrays $T_l$ ($0 \leq l \leq k$) and arrays $B_0$ and $N_0$

| $T_4$ | $T_3$ | $T_2$ | $T_1$ | $T_0$ |
|------|------|------|------|------|
| $\$$ | $T$  | $C$  | $G$  | $T$  |
| $\$$ | $A$  | $C$  | $C$  | $T$  |
| $\$$ | $A$  | $A$  | $C$  | $A$  |

$B_0 = \langle T, T, A \rangle$

$N_0 = \langle 1, 2, 3 \rangle$

(b) Computing the array $N_1$ (lines 7-11)

\[
\begin{array}{|c|}
\hline
i = 1 \\
Read \ T \ from \ B_0 \\
Read \ 1 \ from \ N_0 \\
Append \ 1 \ to \ \mathcal{P}(T) \\
\hline
\end{array}
\]

$\mathcal{P}($) = $\langle \rangle$

$\mathcal{P}(A) = \langle \rangle$

$\mathcal{P}(C) = \langle \rangle$

$\mathcal{P}(G) = \langle \rangle$

$\mathcal{P}(T) = \langle 1 \rangle$

\[
\begin{array}{|c|}
\hline
i = 2 \\
Read \ T \ from \ B_0 \\
Read \ 2 \ from \ N_0 \\
Append \ 2 \ to \ \mathcal{P}(T) \\
\hline
\end{array}
\]

$\mathcal{P}($) = $\langle \rangle$

$\mathcal{P}(A) = \langle \rangle$

$\mathcal{P}(C) = \langle \rangle$

$\mathcal{P}(G) = \langle \rangle$

$\mathcal{P}(T) = \langle 1, 2 \rangle$

\[
\begin{array}{|c|}
\hline
i = 3 \\
Read \ A \ from \ B_0 \\
Read \ 3 \ from \ N_0 \\
Append \ 3 \ to \ \mathcal{P}(A) \\
\hline
\end{array}
\]

$\mathcal{P}($) = $\langle \rangle$

$\mathcal{P}(A) = \langle 3 \rangle$

$\mathcal{P}(C) = \langle \rangle$

$\mathcal{P}(G) = \langle \rangle$

$\mathcal{P}(T) = \langle 1, 2 \rangle$

$N_1 \leftarrow \mathcal{P}(A) \mathcal{P}(T)$ (line 11)

$N_1 = \langle 3, 1, 2 \rangle$

(c) Computing the array $B_1$ (lines 12-14)

\[
\begin{array}{|c|}
\hline
i = 1 \\
Read \ 3 \ from \ N_1 \\
B_1[1] \leftarrow T_1[3] \\
\hline
\end{array}
\]

$B_1 = \langle C \rangle$

\[
\begin{array}{|c|}
\hline
i = 2 \\
Read \ 1 \ from \ N_1 \\
B_1[2] \leftarrow T_1[1] \\
\hline
\end{array}
\]

$B_1 = \langle C, G \rangle$

\[
\begin{array}{|c|}
\hline
i = 3 \\
Read \ 2 \ from \ N_1 \\
B_1[3] \leftarrow T_1[2] \\
\hline
\end{array}
\]

$B_1 = \langle C, G, C \rangle$

Figure 4: Example of iteration $l = 1$ (computing $B_1$ and $N_1$ from $B_0$ and $N_0$) of Algorithm 2 for the set of reads presented in Figure 2. Observe that lists \mathcal{P}($)\mathcal{P}(C), \mathcal{P}(G) are empty during (and at the end of) the iterations of lines 7-10. Angle brackets are used for denoting both lists \mathcal{P}($) and arrays $B_0, B_1, N_0, N_1$. Indeed the latter four can be treated as lists since they are accessed sequentially.
Algorithm 2: Compute the partial BWTs $B_0, B_1, \ldots, B_k$

**Input**: The arrays $T_0, \ldots, T_k$

1. for $i \leftarrow 1$ to $m$
   2. $B_0[i] \leftarrow T_0[i]$;
   3. $N_0[i] \leftarrow i$;
4. for $l \leftarrow 1$ to $k$
   5. for $h \leftarrow 0$ to $\sigma$
      6. $\mathcal{P}(c_h) \leftarrow$ empty list;
   7. for $i \leftarrow 1$ to $m$
      8. $c_h \leftarrow B_{l-1}[i]$;
      9. $q \leftarrow N_{l-1}[i]$;
     10. Append $q$ to $\mathcal{P}(c_h)$;
    11. $N_l \leftarrow \mathcal{P}(c_0)\mathcal{P}(c_1)\ldots\mathcal{P}(c_\sigma)$;
8. for $i \leftarrow 1$ to $m$
   9. $q \leftarrow N_l[i]$;
   10. $B_l[i] \leftarrow T_l[q]$;

$X_0, X_1, \ldots, X_k$, giving the lexicographic ordering of all suffixes of $S$, is equal to the encoding $I_B$ of the interleave of the partial BWTs $B_0, B_1, \ldots, B_k$ giving the BWT $B$, hence we will describe how to compute the encoding $I_X$, from which to reconstruct the BWT $B$. Since the main idea of our algorithm is inspired from radix sort, that is to iteratively perform scans on the set of suffixes such that the overall ordering of the suffixes converges to the lexicographical ordering of them, we must introduce an order relation that will be instrumental in describing the algorithm and proving its correctness.

**Definition 2.** Let $\alpha = s_{i_\alpha}[k - l_\alpha + 1 :]$ and $\beta = s_{i_\beta}[k - l_\beta + 1 :]$ be two generic suffixes of $S$, with length respectively $l_\alpha$ and $l_\beta$, and let $p$ be an integer. Then $\alpha \prec_p \beta$ (and we say that $\alpha$ $p$-precedes $\beta$) iff one of the following conditions holds: (1) $\alpha[: p]$ is lexicographically strictly smaller than $\beta[: p]$, (2) $\alpha[: p] = \beta[: p]$ and $l_\alpha < l_\beta$, (3) $\alpha[: p] = \beta[: p]$, $l_\alpha = l_\beta$ and $i_\alpha < i_\beta$.

The previous definition will be fundamental in our method and is used in the following definition of $p$-interleave.

**Definition 3 ($p$-interleave).** Given the arrays $X_0, X_1, \ldots, X_k$, the $p$-interleave $X_p$ ($0 \leq p \leq k$) is the interleave of $X_0, X_1, \ldots, X_k$ such that $X_p[i]$ is the $i$-th smallest suffix in the $\prec_p$-ordering of all the suffixes in $X_0 \cup X_1 \cup \ldots \cup X_k$.

Observe that $\prec_\infty$ is the usual lexicographic order and it is immediate to verify that $\prec_k$ corresponds to $\prec_\infty$, and therefore $X^k$ (that is, the suffixes sorted according to the $\prec_k$ relation) is equal to $X$, hence $I_X = I_{X^k}$. The ordering $\prec_0$ is trivial, since $X^0$ is the concatenation of arrays $X_0, X_1, \ldots, X_k$ and the encoding $I_{X^0}$ is given by $m$ 0s, followed by $m$ 1s, $\ldots$, followed by $m$ values equal to $k$. 

Our procedure is iterative and computes \( I_{X^k} \) starting from \( I_{X^0} \); the iteration \( p \) computes \( I_{X^{p+1}} \) from \( I_{X^p} \), and the details are shown in Algorithm 3.

To build \( I_{X^{p+1}} \) from \( I_{X^p} \), Algorithm 3 builds a set of \( \sigma + 1 \) lists \( \mathcal{I}(c_0), \mathcal{I}(c_1), \ldots, \mathcal{I}(c_\sigma) \) that is the partitioning of the elements of \( I_{X^{p+1}} \) by the first character \( c_i \) \( (0 \leq i \leq \sigma) \) of their related suffixes. Since the list \( \mathcal{I}(c_0 = \$) \) is related to the empty suffixes, it is fixed (by Definition 2) over the iterations and is always composed of \( m \) 0s. Finally, the algorithm produces \( I_{X^{p+1}} \) (see line 8) as the concatenation \( \mathcal{I}(c_0)\mathcal{I}(c_1) \cdots \mathcal{I}(c_\sigma) \).

The ordering given by \( X^{p+1} \) from \( X^p \) is performed by considering the symbols preceding the suffixes of \( X^p \) and reordering the suffixes by that symbols. With respect to the encoding \( I_{X_p} \), this is translated into the instructions at line 3 of Algorithm 3 where the algorithm performs a scan of \( I_{X^p} \). For each position \( i \), it obtains \( l = I_{X^p}[i] \), that is the length of the \( i \)-th suffix in the \( \prec_p \)-ordering, and the symbol \( c \) preceding such suffix (see lines 1-3). If \( c \neq \$ \), then \( l < k \), and \( l+1 \) is appended to the list \( \mathcal{I}(c) \). Otherwise, if \( c = c_0 = \$ \), it moves to the next position \( i+1 \). Indeed, in this case the suffix related to position \( i \) of \( I_{X^p} \) has length \( k \) and the ordering of strings in \( X_0 \) corresponds to the ordering related to \( \mathcal{I}(c_0) \) which is fixed.

In Figure 5 the computation of \( I_{X^1} \) from \( I_{X^0} \) is shown for the set \( S \) of reads presented in Figure 2. The interleaves \( I_{X^1} \) and \( I_{X^0} \) and their relation with the suffixes of \( S \) is reported in Figure 6.

The following theorem proves the correctness of Algorithm 3.

**Theorem 2.** If Algorithm 3 receives in input the encoding \( I_{X^p} \) of the \( p \)-interleave \( X^p \), then it computes the encoding \( I_{X^{p+1}} \) of the \((p+1)\)-interleave \( X^{p+1} \).

**Proof.** The proof is directly based on Definition 2 of \( \prec_p \)-ordering. By hypothesis \( X^p \) consists of the \( p \)-interleave of \( X_0, X_1, \ldots, X_k \), that is, the suffixes in \( X_0 \cup X_1 \cup \ldots \cup X_k \) are sorted w.r.t. the \( \prec_p \)-ordering. Observe that, for each \( l \), suffixes in \( X_l \) follows a \( \prec_p \)-ordering for any \( p \). By Definition 2 the \( \prec_{p+1} \)-ordering of suffixes in \( X_0 \cup X_1 \cup \ldots \cup X_k \) is obtained by sorting w.r.t. prefixes of length \( p+1 \), that is, given two suffixes \( w_i \) and \( w_j \), then \( w_i \prec_{p+1} w_j \) if one the following conditions hold: (1) \( w_i[: p+1] \) is lexicographically smaller than \( w_j[: p+1] \), or (2) \( w_i[: p+1] = w_j[: p+1] \), \( w_i \in X_{l_i} \), \( w_j \in X_{l_j} \) and \( l_i < l_j \) or (3) \( w_i[: p+1] = w_j[: p+1] \), \( w_i \in X_l \), and the rank of \( w_i \) in \( X_l \) is smaller than the one of \( w_j \) in \( X_l \).

Observe that the \((p+1)\)-prefix of the \( i \)-th suffix \( w_i \) in \( X_l \) is the \((p+1)\)-prefix of a suffix \( cw_i \) in \( X_{l+1} \), where \( c = B_i[i] \), since \( c \) is the symbol that precedes \( w_i \) in a string \( s \) in \( S \). Then, line 7 of Algorithm 3 applies length \( l + 1 \) to the list \( \mathcal{I}(c) \). Observe that line 7 implicitly computes a partitioning of the suffixes in \( X^{p+1} \), according to their starting symbol \( c_i \), into lists \( L(c_0), L(c_1), \ldots, L(c_\sigma) \), being \( L(c_i) \) the \( \prec_{p+1} \)-ordering of suffixes starting with symbol \( c_i \). The list \( \mathcal{I}(c_i) \) contains (at line 8) the lengths of the suffixes in \( L(c_i) \).

By definition of \( \prec_{p+1} \)-ordering, given two distinct suffixes \( c_1 w_i \) and \( c_2 w_j \) such that \( c_1 w_i \prec_{p+1} c_2 w_j \), either they begin with two different symbols \( c_1 < c_2 \), or they both starts with the same symbol, i.e., \( c_1 = c_2 \). Given \( L(c_1) \) and \( L(c_2) \) the lists of suffixes of \( X^p \) that start with \( c_1 \) and \( c_2 \) (respectively), then in \( X^{p+1} \) all suffixes in \( L(c_1) \) precedes those in
(a) Input interleave $I_{X^0}$

$I_{X^0} = \langle 0, 0, 0, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4 \rangle$

(b) Initialization of lists $I(\cdot)$

$I($) = \langle 0, 0, 0 \rangle$
$I(A) = \langle \rangle$
$I(C) = \langle \rangle$
$I(G) = \langle \rangle$
$I(T) = \langle \rangle$

(c) Scan of the interleave $I_{X^0}$ (lines 3-7)

\[
\begin{array}{ccc}
\hline
i & \text{Read} & \text{Append} \\
\hline
1 & 0 & \text{I(T)} \\
2 & 0 & \text{I(T)} \\
3 & 1 & \text{I(A)} \\
4 & 1 & \text{I(T)} \\
5 & 1 & \text{I(A)} \\
6 & 1 & \text{I(C)} \\
7 & 2 & \text{I(A)} \\
8 & 2 & \text{I(C)} \\
9 & 2 & \text{I(C)} \\
10 & 3 & \text{I(A)} \\
11 & 3 & \text{I(A)} \\
12 & 3 & \text{I(T)} \\
13 & 4 & \text{I(A)} \\
14 & 4 & \text{I(A)} \\
15 & 4 & \text{I(T)} \\
\hline
\end{array}
\]

$I($) = \langle 0, 0, 0 \rangle$
$I(A) = \langle 1, 3, 4, 4 \rangle$
$I(C) = \langle 2, 2, 3, 3 \rangle$
$I(G) = \langle 2 \rangle$
$I(T) = \langle 1, 1, 4 \rangle$

(d) Computing the interleave $I_{X^1}$ (line 8)

$I_{X^1} \leftarrow I($)I(A)I(C)I(G)I(T)$

$I_{X^1} = \langle 0, 0, 0, 1, 3, 4, 4, 2, 2, 3, 3, 2, 1, 1, 4 \rangle$

Figure 5: Example of computing $I_{X^1}$ from $I_{X^0}$ (see Algorithm 3) for the set of reads presented in Figure 2. Angle brackets are used for denoting both lists $I(\cdot)$ and arrays $I_{X^0}$, $I_{X^1}$. Indeed the latter two can be treated as lists since they are accessed sequentially.
Definition 4. Given the LCP array, \( LCP_p \) is defined such that \( LCP_p[i] = \min\{LCP[i], p\} \).

Observe that \( LCP_p[i] \) is the length of the longest prefix shared by the \( p \)-prefix of \( X^p[i] \) and the \( p \)-prefix of \( X^p[i − 1] \). The array \( LCP_k \) is equal to the LCP array of the input set \( S \), and \( LCP_0 \) contains all 0s, except for \( LCP_0[1] \) that is equal to −1. In Figure 6, \( LCP_0 \) and \( LCP_1 \) are reported for the input set of Figure 2.

Our procedure iteratively computes \( LCP_k \) starting from \( LCP_0 \); the iteration \( p \) computes \( LCP_{p+1} \) from \( LCP_p \). Algorithm 4 extends Algorithm 3 to compute both \( I_{X^{p+1}} \) and \( LCP_{p+1} \) from \( I_{X^p} \) and \( LCP_p \).

Algorithm 3: Compute \( I_{X^{p+1}} \) from \( I_{X^p} \)

1. \( I(c_0) \leftarrow 0, 0, \ldots, 0; \)
2. \( I(c_1), \ldots, I(c_\sigma) \leftarrow \) empty lists;
3. for \( i \leftarrow 1 \) to \( |I_{X^p}| \) do
   4. \( l \leftarrow I_{X^p}[i]; \)
   5. \( c \leftarrow B_l[\text{rank}_{I_{X^p}}(i, l)]; \)
   6. if \( c \neq \$ \) then
      7. Append \( l + 1 \) to \( I(c); \)
8. \( I_{X^{p+1}} \leftarrow I(c_0)I(c_1)\cdots I(c_\sigma); \)

As stated before, in the following we will describe how to compute the LCP array of the input dataset. Similarly to the computation of the BWT \( B \), the LCP array will be constructed iteratively. More precisely, the LCP array will be constructed by considering prefixes of the suffixes by increasing length.

The following definition will be fundamental in the following.

\( L(c_2) \). Inside the list \( L(c_1) \), the \( \prec_{p+1} \)-ordering of \( c_1w_i \) and \( c_1w_i \) is the same as \( X^p \). Indeed, \( cw_i[:p] \) is lexicographically smaller than \( cw_j[:p] \) if and only if \( w_i[:p] \) is lexicographically smaller than \( w_j[:p] \). It follows that \( X^{p+1} \) consists of the concatenation of \( L(c_i) \) according to the lexicographic ordering of symbols of alphabet \( \Sigma \), and thus line 8 of Algorithm 3 computes the encoding \( I_{X^{p+1}} \) of \( X^{p+1} \).

Before giving some more detail, we need to introduce the following function. Given a position \( i \) and a symbol \( c \neq \$ \), the function \( \alpha_p(i, c) \) is the length of the longest prefix shared by the \( p \)-prefixes of \( X^p[i] \) and \( X^p[h] \) where \( h \) is the largest integer such that \( h < i \) and \( X^p[h] \) is preceded by \( c \). In the following, given two strings \( x_1, x_2 \), by \( lcp_p(x_1, x_2) \) and \( lcp(x_1, x_2) \) we denote (respectively) the length of the longest common prefix between the \( p \)-prefix of \( x_1 \) and \( x_2 \), and the length of the longest common prefix between \( x_1 \) and \( x_2 \).
(that is $lcp(x_1, x_2) = lcp_k(x_1, x_2)$). If no such $h$ exists, then $\alpha_p(i, c) = -1$. The following proposition relates the values of $\alpha_p(i, c)$ and the array $LCP_{p+1}$ and it is a direct consequence of their definitions.

**Proposition 3.** Let $cw_1$ and $cw_2$ be two consecutive suffixes in $X^{p+1}$ w.r.t. the $\prec_{p+1}$ ordering, and let $w_2$ be the $i$-th suffix w.r.t. the $\prec_p$ ordering. Then $\min\{p + 1, lcp(cw_1, cw_2)\} = 1 + \alpha_p(i, c)$.

During the scan of the encoding $I_{X^p}$, the value $LCP_p[i]$ is computed (see line 13 of Algorithm 4). The function $\alpha_p(i, c)$ is maintained in the array $\alpha$ of size $\sigma - 1$ initially set to $\sigma - 1$ values $-1$s (see line 7), and updated in the cycle at line 14. The main invariant of Algorithm 4 is that, at line 16, the variable $\alpha[c]$ is equal to $\alpha_p(i, c)$ — this invariant is a consequence of the following Lemma 4 and can be proved by a direct inspection of Algorithm 4. The value $\alpha[c]$ incremented by 1 is appended to the list $L(c)$.

**Lemma 4.** Let $w_1$ and $w_2$ be respectively the $j$-th and the $i$-th suffixes w.r.t. the $\prec_j$ ordering, with $j < i$, and let $c$ be the symbol preceding suffix $w_1$. If every suffix of position $t$ w.r.t. the $\prec_j$ ordering with $j < t < i$ is not preceded by the symbol $c$, then it holds that $\alpha_p(i, c) = \min_{j < h \leq i}\{LCP_p[h]\}$.

**Proof.** Since $c$ is not the symbol that precedes the suffix of position $t$ w.r.t. the $\prec_j$ ordering with $j < t < i$, then by definition of $\alpha_p(i, c)$, it must be that $\alpha_p(i, c) = lcp_p(X^p[j], X^p[i])$, since the $j$ is the largest integer less than $i$ for which the $j$-th suffix is preceded by symbol $c$. Since it is immediate to verify that $lcp_p(X^p[j], X^p[i]) = \min_{j < h \leq i}\{LCP_p[h]\}$, the lemma easily follows.

The previous argument allows to prove the following theorem which, combined with Theorem 2, completes the correctness of Algorithm 4.

**Theorem 5.** Given as input $LCP_p$ and the partial BWTs $B_0, B_1, \cdots, B_k$, Algorithm 4 computes the list $LCP_{p+1}$.

**Proof.** Notice that $\alpha[c] \geq 0$ at line 16 iff the current suffix at position $i$ is not the first to be preceded by the character $c$, hence we must append the value $1 + \alpha_p(i, c)$ to $L(c)$. Since $\alpha[c] = \alpha_p(i, c)$, the theorem is proved.

The procedure $\text{BWT}+\text{LCP}$ (see Algorithm 5) computes $I_{X^k}$ and $LCP_k$, which are the encoding of the BWT and the LCP array of the input set $S$ of reads, by iterating Algorithm 4. Iterations stop when the maximum value $\max_i\{LCP_p[i]\}$ in the array $LCP_p$ is less than $p$. In fact, it means that for an iteration $t$ of the algorithm with $t$ larger than $p$, the values $I_{X^t}$ and $LCP_t$ do not change as the suffixes have been fully sorted and thus they remain equal to $I_{X^k}$ and $LCP_k$, respectively. The correctness of the procedure $\text{BWT}+\text{LCP}$ is a consequence of Theorem 5 and Definition 4. Observe that if the maximum value in the LCP array is equal to $z$, then at each $p$-iteration of Algorithm 5 with $p <= z$, the maximum value in $LCP_p$ is $p$, in virtue of Theorem 5 and Definition 4. When $p = z + 1$, then by Definition 4, the $p$-iteration gives value $z$, that is $\max_i\{LCP_p[i]\} < p$. Then the suffixes have been fully sorted and the LCP array has been computed at the previous step $p = z$. 

13
Algorithm 4: Compute $I_{X^{p+1}}$ and $LCP_{p+1}$ from $I_{X^p}$ and $LCP_p$

1. $\mathcal{I}(c_0) \leftarrow 0, 0, \ldots, 0$;
2. $\mathcal{I}(c_1), \ldots, \mathcal{I}(c_\sigma) \leftarrow$ empty lists;
3. $\mathcal{L}(c_0) \leftarrow -1, 0, \ldots, 0$;
4. $\mathcal{L}(c_1), \ldots, \mathcal{L}(c_\sigma) \leftarrow$ empty lists;
5. foreach $c \in \{c_1, \ldots, c_\sigma\}$ do
6.     $\mathcal{L}(c) \leftarrow$ the list $<0>$;
7.     $\alpha \leftarrow \{-1, -1, \ldots, -1\}$;
8. for $i \leftarrow 1$ to $|I_{X^p}|$ do
9.     $l \leftarrow I_{X^p}[i]$;
10.    $c \leftarrow B_l[\text{rank}_{I_{X^p}}(i, l)]$; // is the character preceding the current suffix
11.    if $c \neq \$ then
12.        Append $(l + 1)$ to $\mathcal{I}(c)$;
13.        $lcp \leftarrow LCP_p[i]$;
14.        foreach $d \in \{c_1, \ldots, c_\sigma\}$ do
15.            $\alpha[d] \leftarrow \min\{\alpha[d], lcp\}$;
16.            if $c \neq \$ and $\alpha[c] \geq 0$ then
17.                Append $\alpha[c] + 1$ to $\mathcal{L}(c)$;
18.                $\alpha[c] = \infty$;
19.     $I_{X^{p+1}} \leftarrow \mathcal{I}(c_0)\mathcal{I}(c_1)\cdots\mathcal{I}(c_\sigma)$;
20.    $LCP_{p+1} \leftarrow \mathcal{L}(c_0)\mathcal{L}(c_1)\cdots\mathcal{L}(c_\sigma)$;

Algorithm 5: BWT+LCP

Input : The strings $s_1, \ldots, s_m$, each $k$ long
Output : The BWT $B$ and the LCP array of the input strings
1. Compute $T_0, \ldots, T_k$ from $s_1, \ldots, s_m$;
2. Apply Algorithm 2 to compute $B_0, \ldots B_k$ from $T_0, \ldots, T_k$;
3. $I_{X^0} \leftarrow m 0s, m 1s, \ldots, m ks$;
4. $LCP_0 \leftarrow -1, 0, 0, \ldots, 0$;
5. $p \leftarrow 0$;
6. while $\max_i \{LCP_{p+1}[i]\} = p + 1$ do
7.    Apply Algorithm 4 to compute $I_{X^{p+1}}$ and $LCP_{p+1}$ from $I_{X^p}$, $LCP_p$ and the lists $B_0, \ldots, B_k$;
8.    $p \leftarrow p + 1$;
9. Reconstruct $B$ from $I_{X^p}$ and $B_0, \cdots, B_k$;

Output : $(B, LCP_p)$

6 Complexity

In this section we will analyze the computational and I/O complexity of our algorithm, comparing them with those of extLCP.
Table 3: Interleaves $I_{X^0}$ and $I_{X^1}$, and arrays $LCP_0$ and $LCP_1$ for the set of reads presented in Figure 2.

| $I_{X^0}$ | $LCP_0$ | $X^0$ | $I_{X^1}$ | $LCP_1$ | $X^1$ |
|-----------|---------|-------|-----------|---------|-------|
| 0         | 1       | $\$$ | 0         | 1       | $\$$ |
| 0         | 0       | $\$$ | 0         | 0       | $\$$ |
| 0         | 0       | $\$$ | 0         | 0       | $\$$ |
| 1         | 0       | A$\$$| 1         | 0       | A$\$$|
| 1         | 0       | T$\$$| 3         | 1       | ACAS$|
| 1         | 0       | T$\$$| 4         | 1       | AACA$|
| 2         | 0       | CA$\$$| 4         | 1       | ACCT$|
| 2         | 0       | CT$\$$| 2         | 0       | CA$\$$|
| 2         | 0       | GT$\$$| 2         | 1       | CT$\$$|
| 3         | 0       | ACA$\$$| 3         | 1       | CCT$\$$|
| 3         | 0       | CCT$\$$| 3         | 1       | CGT$\$$|
| 3         | 0       | CGT$\$$| 2         | 0       | GT$\$$|
| 4         | 0       | AACA$\$$| 1         | 0       | T$\$$|
| 4         | 0       | ACCT$\$$| 1         | 1       | T$\$$|
| 4         | 0       | TCGT$\$$| 4         | 1       | TCGT$\$$|

Figure 6: Interleaves $I_{X^0}$ and $I_{X^1}$, and arrays $LCP_0$ and $LCP_1$ for the set of reads presented in Figure 2.

First we will analyze Algorithm 2. This procedure mainly consists of two nested loops in which each operation requires constant time. If the input is a set of $m$ strings of length $k$, the time complexity of it is $O(mk)$. Note that each of the $k+1$ lists $B_i$ and $N_i$ have $m$ elements which are read or written sequentially and, moreover, each list is read only once per execution. Hence, the I/O complexity of Algorithm 2 is $O(mk \log m)$ since, for each element in $T_0, \ldots, T_i$, Algorithm 2 appends an integer $q \leq m$ to the correct list $P(\cdot)$ that we can store on disk, since we access them sequentially.

Besides some $O(1)$-space data structures, the algorithm uses $\sigma + 1$ lists $P(\cdot)$ to store pointers to the files and $k + 1$ arrays $T_i$ to store the characters of the sequences. Moreover, as stated in Section 4, this algorithm requires to maintain a vector $T_i$ of sequence indexes in main memory due to random access. At each iteration of the loop at lines 12–14, an array $T_i$ must be kept in main memory, since we need to perform non-sequential accesses. Therefore, if we can address each file using $w$ bits, the main memory requirement of Algorithm 2 is $O(\sigma w + kw + m \log \sigma)$ bits.

We will now analyze Algorithm 4.

The time complexity of this procedure is $O(mk\sigma)$ since such procedure is composed of a for loop that iterates over the encoding of the interleave $I_{X^P}$ — whose length is $mk$ — performing constant time operations per element except for the loop at lines 14–15 that requires $O(\sigma)$ time.

The I/O complexity is $O(mk \max\{\log m, \log l\})$ bits, since each iteration of the loop at lines 14–15 requires to read and write a constant number of elements of some lists whose values are bounded by $m$ or $l$, and since $\alpha$ is not considered in this analysis since it’s kept
in main memory. The main memory usage is $O(\sigma \lg l + kw)$ bits, since we store $\sigma$ integers smaller than $l$ and $k$ pointers to the lists $B_i$.

We can now analyze Algorithm 5, which is composed of two main steps: in the first one it prepares the input data structures (line 1), invokes Algorithm 2, and initializes some data structures. In the second part (lines 6–8) it computes the final encoding of the interleave $I_{X^p}$ and the LCP array from the structures computed at the previous step by iteratively applying Algorithm 4.

The complexity of the first part is essentially that of Algorithm 2, since computing the lists $T_0, \ldots, T_k$ (line 1) requires $O(mk)$. In fact, it requires a single scan of the input data (whose size is $mk$), while outputting the lists requires constant time per element.

The second step of is mainly composed of a while loop that iteratively applies Algorithm 4 to compute the final interleave and the final LCP array. Moreover, the proof of correctness of Algorithm 5 also shows that Algorithm 4 is applied $l + 1$ times, where $l$ is the largest value in the LCP array.

Finally, Algorithm 5 builds the final BWT from $I_{X^p}$ and the lists $B_0, \ldots, B_k$ by a single scan of those $O(mk)$-long lists, which requires $O(mk)$ time. Therefore, Algorithm 5 requires an overall $O(mkl\sigma)$ time.

The I/O complexity of the first step is $O(\max\{mk\lg m, mk\lg \sigma\})$ bits whereas the main memory requirement is $O(m\lg m)$ bits. Indeed, computing the lists $T_0, \ldots, T_k$ at line 1 requires us to store only one character per time of each sequence $s_i$ and to append it to the correct list: therefore it has $O(mk\lg \sigma)$ bits I/O complexity and $O(kw + \lg \sigma)$ bits main memory requirement. We have to include the requirements of Algorithm 2 which changes the main memory needed for the first step to $O(\sigma w + kw + m\lg \sigma)$ bits.

The I/O complexity of the second step is $O(mkl\lg l)$ bits since it consists essentially of $l$ applications of Algorithm 4. Finally, while building the final BWT from $I_{X^p}$, Algorithm 5 reads $O(mk\lg m)$ bits due to the interleave and $O(mk\lg \sigma)$ bits due to the partial BWTs, writes $O(mk\lg \sigma)$ bits for the final BWT and requires $O(\max\{\lg l, \lg \sigma\})$ bits of main memory since at most it stores in main memory one element of $I_{X^p}$ and one element of a partial BWT.

Therefore, overall Algorithm 5 reads and writes $O(mkl \max\{\lg m, \lg l\})$ bits from and to the disk and requires $O(\sigma w + kw + m\lg \sigma + \lg l)$ bits of main memory. We can summarize our results as follows.

**Proposition 6.** Given as input a set composed of $m$ strings of length $k$ over and alphabet of size $\sigma$, the procedure BWT+LCP computes the BWT and the LCP array of it in $O(mkl\sigma)$ time, where $l$ is the maximal value of the LCP array. This procedure requires to store in main memory $O(\sigma w + kw + m\lg \sigma + \lg l)$ bits and reads and writes from and to the disk $O(mkl \max\{\lg m, \lg l\})$ bits.

Note that, if $\sigma$ is constant then the time complexity of the method presented in this paper becomes $O(mkl)$. Moreover, if the word size is $\max\{w, \lg m, \lg l\}$ then its I/O complexity and main memory requirement become $O(mkl)$ and $O(k + m)$ respectively.
7 Conclusions

We have presented a new lightweight algorithm to compute the BWT and the LCP array of a set of \( m \) strings, each one \( k \) characters long. More precisely, our algorithm has an \( O(mkl) \) time and I/O complexity, and uses \( O(k + m) \) main memory to compute the BWT and LCP array, where \( l \) is the maximum value in the LCP array. Our time and I/O complexity are at least as good as those of extLCP\([8]\), the best available algorithm.

While our focus has been on the theoretical aspects, it would be interesting to implement the proposed algorithm and perform an experimental analysis to determine the practical behavior, especially because our approach is entirely based on linear scans.

Acknowledgements

The authors acknowledge the support of the MIUR PRIN 2010-2011 grant “Automi e Linguaggi Formali: Aspetti Matematici e Applicativi” code 2010LYA9RH, of the Cariplo Foundation grant 2013-0955 (Modulation of anti cancer immune response by regulatory non-coding RNAs).

References

[1] Markus Bauer, Anthony Cox, and Giovanna Rosone. Lightweight BWT construction for very large string collections. In Combinatorial Pattern Matching, volume 6661 of LNCS, pages 219–231. Springer, 2011.

[2] Markus Bauer, Anthony Cox, and Giovanna Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. Theoretical Computer Science, 483:134–148, 2013.

[3] Markus Bauer, Anthony Cox, Giovanna Rosone, and Marinella Sciortino. Lightweight LCP construction for next-generation sequencing datasets. In Algorithms in Bioinformatics, volume 7534 of LNCS, pages 326–337, Berlin, Germany, 2012. Springer.

[4] 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.

[5] Paola Bonizzoni, Gianluca Della Vedova, Serena Nicosia, Marco Previtali, and Raffaella Rizzi. A new lightweight algorithm to compute the bwt and the lcp array of a set of strings. CoRR, abs/1607.08342, 2016.

[6] Paola Bonizzoni, Gianluca Della Vedova, Yuri Pirola, Marco Previtali, and Raffaella Rizzi. LSG: an external-memory tool to compute string graphs for next-generation sequencing data assembly. Journal of Computational Biology, 23(3):137–149, 2016.
[7] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Systems Research Center, 1994.

[8] Anthony Cox, Fabio Garofalo, Giovanna Rosone, and Marinella Sciortino. Lightweight lcp construction for very large collections of strings. *Journal of Discrete Algorithms*, 37(C):17–33, March 2016.

[9] Paolo Ferragina, Travis Gagie, and Giovanni Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730, 2012.

[10] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. of the ACM*, 52(4):552–581, 2005.

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

[12] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10:R25, 2009.

[13] Heng Li. Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly. *Bioinformatics*, 28(14):1838–1844, July 2012.

[14] Heng Li. Fast construction of FM-index for long sequence reads. *Bioinformatics*, 30(22):3274–3275, 2014.

[15] Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 15:1754–1760, 2009.

[16] Heng Li and Richard Durbin. Fast and accurate long-read alignment with burrows–wheeler transform. *Bioinformatics*, 26:589–595, 2010.

[17] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. Soap2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25:1966–1967, 2009.

[18] Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the burrows wheeler transform and applications to sequence comparison and data compression. In Alberto Apostolico, Maxime Crochemore, and Kunsoo Park, editors, *Combinatorial Pattern Matching*, volume 3537 of *Lecture Notes in Computer Science*, pages 178–189. Springer Berlin Heidelberg, 2005.

[19] Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the burrows-wheeler transform. *Theoretical Computer Science*, 387:298–312, 2007.
[20] Eugene Myers. The fragment assembly string graph. *Bioinformatics*, 21(suppl. 2):ii79–ii85, 2005.

[21] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *Data Compression Conference, 2009. DCC’09.*, pages 193–202. IEEE, 2009.

[22] Junjie Qin, Ruiqiang Li, Jeroen Raes, Manimozhiyan Arumugam, Kristoffer Solvsten Burgdorf, Chaysavanh Manichanh, Trine Nielsen, Nicolas Pons, Florence Levenez, Takuji Yamada, et al. A human gut microbial gene catalogue established by metagenomic sequencing. *Nature*, 464(7285):59–65, 2010.

[23] Giovanna Rosone and Marinella Sciortino. The Burrows-Wheeler transform between data compression and combinatorics on words. In *The Nature of Computation. Logic, Algorithms, Applications*, volume 7921 of *LNCS*, pages 353–364. Springer Berlin Heidelberg, 2013.

[24] Jared Simpson and Richard Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinformatics*, 26(12):i367–i373, 2010.

[25] Jared Simpson and Richard Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome Research*, 22:549–556, 2012.

[26] Jouni Sirén. Compressed suffix arrays for massive data. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval, SPIRE ’09*, pages 63–74, Berlin, Heidelberg, 2009. Springer-Verlag.

[27] Niko Välimäki, Susana Ladra, and Veli Mäkinen. Approximate all-pairs suffix/prefix overlaps. In Amihood Amir and Laxmi Parida, editors, *Combinatorial Pattern Matching*, volume 6129 of *Lecture Notes in Computer Science*, pages 76–87. Springer Berlin Heidelberg, 2010.