Università degli Studi di Milano Bicocca
Scuola di Scienze
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di laurea in Informatica

Algorithm Engineering: an algorithm for BWT and LCP

Relatore: *Prof. Gianluca Della Vedova*
Co-relatore: *Dott. Marco Previtali*

Relazione della prova finale di:
Fabio Nicolini
Matricola 794467

Anno Accademico 2016–2017

Abstract

This thesis describes problems, goals and achievements encountered while implementing an algorithm for computing the Burrows-Wheeler Transform (BWT) and Longest Common Prefix (LCP) exploiting external memory and minimizing the RAM usage.

The requirement of a low RAM usage for this task appears in fields where a large amount of data needs to be processed and manipulated, such as Computational Biology.

The introduction of more accessible DNA sequencing techniques made this constraint more impending: more data, composed of shorter but more numerous reads has highly increased the RAM usage of existent programs.

Contents

1 Introduction

The program discussed in this paper can be considered as a sub-routine of a larger project: a genome assembler implemented with the low RAM usage philosophy. BWT and LCP construction is a fundamental step towards developing of such software. Before discussing in detail the BWT and LCP construction we will introduce a very important aspect of computational biology: the origin of molecular sequence data, since string problems on such data provide a large part of the motivation for studying string algorithms in general.

1.1 DNA sequencing

The purpose of DNA sequencing is to determine the exact order of nucleotides within a DNA molecule. This task is of extreme importance and has application in many different fields: from molecular and evolutionary biology to medicine and forensics. It is in the essentially simple nature of the DNA that lies the fundamental structure of life:

> The digital information that underlies biochemistry, cell biology, and development can be represented by a simple string of G's, A's, T's and C's. This string is the root data structure of an organism's biology. [19]

The meeting point between biology and computer science is the assumption that biologically meaningful results could come from considering the DNA as a one-dimensional character string, abstracting away the reality of DNA as a flexible three-dimensional molecule. To exactly determine this one-dimensional character string is not a simple task. Through exploiting an organism's biochemistry, it is possible to sequence its DNA in different ways, though all these techniques share similar problems; we will examine in detail a technique called *shotgun sequencing*. The main problem is that it's not possible to sequentially "read" many nucleotides, but at most a few hundred.(1000 - 1200 Sanger method, 50-500 NGS) [11] This is makes the sequencing task way more difficult and prompts to look for different approaches. Because of this issue it is not possible to reconstruct the DNA molecule starting from a single copy of such molecule. This is because even by fragmenting a single copy of DNA, there is not enough information to determine the order of the fragments. We will introduce a possible solution to this problem: the *shotgun sequencing protocol*.

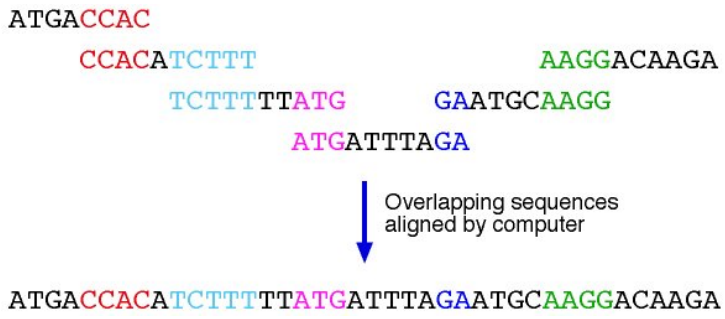

Figure 1: Overlaps between fragments used to reconstruct original string

1.1.1 Shotgun sequencing

Shotgun sequencing is a protocol that has to be used in combination with other techniques that actually perform the reading task(like Sanger or NGS). In fact, shotgun sequencing specifies *how* to use the data read by other technologies in order to sequence large section of DNA. The basic idea of this solution is to use multiple copies of the DNA of interest and then to cut these copies by physical, enzymatic, or chemical means so that each copy is cut in somewhat random locations. The cutting produces fragments from the original DNA, called *reads*. In the fragmentation process the original order of the reads is lost, one simply is left with a set of reads without knowing their initial order in the full string nor from which copy of the string it came from. Still, the reads from different copies overlap each other so that if enough overlapping fragments are sequenced, the common patterns from overlapping sequenced fragments can be used to try assemble the sequence of the full DNA string (see figure). Hence the sequencing problem becomes something very similar to a "jigsaw puzzle" where the basic components are the DNA fragments randomly obtained from the DNA string, and the final picture to discover is the one-dimensional string representation of the DNA. This step is called *sequence assembly*.

1.2 Sequence assembly

The problem of sequence assembly can be compared to taking many copies of a book, passing each of them through a shredder with a different cutter, and piecing the text of the book back together just by looking at the shredded pieces. Besides the obvious difficulty of this task, there are some extra practical issues: the original may have many repeated paragraphs, and

some shreds may be modified during shredding to have typos. Excerpts from another book may also be added in, and some shreds may be completely unrecognizable. [23] It is important to note that the sum of all nucleotides of all reads it's always much higher than the length of the reconstructed sequence. This means that there are always multiple different reads that contain the same nucleotide of the final sequence. Actually, very few of the fragments from the set of reads will be sequenced, since most procedures only sequence fragments of a fixed size range but the fragments' length is somewhat random. So instead of sequencing all the fragments, the shotgun technique randomly chooses fragments from the set of reads of an appropriate length and then it sequences the first X bases, where X is the fixed number used by the procedure (usually around 400 nucleotides). [11]

The number of fragments picked must be sufficient to allow reconstruction of the full DNA string from the overlapping sequenced parts of the fragments. Probabilistic analyses have been made [15] to determine how many fragments to pick. In practice, creation, cloning and selection of fragments is not always uniform enough to allow a complete reconstruction. This is one of the limits of the shotgun protocol, although the issue does not always lie in the probabilistic selection of reads but is due to molecular and biological reasons. In these cases even adding additional coverage does not solve the problem.

Recalling the example of the book reconstruction, we said that there are some extra practical issues. These issues have a huge impact on the problem. For example, the reads may have wrong nucleobases (or missing ones) because the DNA sequencer does not always "read" them correctly. This makes the overlap detection a harder task since, if in reality two given reads should perfectly overlap, they may not because of some errors in the sequencing instrument. Another problem is the presence in the DNA molecule of identical or nearly identical sequences (known as *repeats*). Given a read that is contained, even partially, inside one of these regions it's very difficult to determine the correct position of such read in the DNA string.

Having briefly introduced DNA sequencing and sequence assembly, which comprise real front-line problems, we now step back a bit to introduce a more abstract pure string problem.

1.2.1 Shortest superstring problem

> **Definition** Given a set of string $\Upsilon = \{S_1, S_2, ..., S_k\}$ a *superstring* of the set Υ is a single string that contains every string in Υ as a substring.

For example, a concatenation of the strings of Υ in any order gives a basic superstring of Υ. For a more interesting example, let $\Upsilon = \{actt,\ gtca,\ cttag\}$. Then *cttagacttgtca* is a superstring of Υ, and *gtcaacttag* is another, shorter superstring of Υ. In general we are interested in finding superstrings whose length is small. This is because shorter superstrings take into consideration the overlaps between substrings and tend to unify them as one in the superstring. This is the way to go since, as explained before, overlapping sequences are likely to be in sequential ordering in the final superstring.

The problem of finding the shortest superstring is mostly motivated by one approach to the sequence assembly problem in shotgun sequencing, although it may have other applications such as in data compression. The strings in the set Υ model the sequenced DNA fragments created by the shotgun sequencing protocol and the objective is to deduce the originating DNA string S from the set of sequenced fragments Υ. Without sequencing errors, the full DNA string S is a superstring of Υ and, under the assumptions stated above, S is likely to be a shortest superstring of Υ. Though most of the time S will not be a very "good" string, as the model suggests. In fact, real sequence data indicates that the shortest superstring is often shorter than the originating string S, especially in presence of *repeats*. Therefore the shortest superstring problem is an abstract algorithm that only loosely models the real sequence assembly task. Nonetheless it remains important to study abstract approaches to practical problems because, more often than not, in the techniques and solutions to an abstract problem lies the key to obtain practical results in more complex or realistic systems.

The shortest superstring problem is known to be NP-hard [10], therefore a polynomial-time algorithm to find the exact optimal is thought to be unlikely. Moreover, it has been found that the problem belongs to the MAX-SNP-hard complexity class [4], and so it is also thought that there is no polynomial-time algorithm that can approximate the optimal time within an *arbitrary, but predetermined* constant. However, several linear approximation have been proposed. The first by Blum et al. [4] guarantees an approximation factor of three. The factor has been successively improved many times; the best improvement to date, by Sweedyk[20] and Kaplan et al. separately[14], re-

duces the bound to 2.5. Nevertheless this is still an open problem because of the conjecture called "The Greedy Conjecture" that states: *The Greedy Algorithm for the shortest superstring problem has an approximation factor of 2;* even though such an algorithm has not yet been discovered.

A related problem that has interesting connection with the biological structure of the DNA molecule is the shortest superstring of a collection of strings where each string in the collection can be considered in both forward and backward orientation. This problem is motivated by the fact that most DNA molecules consist of two *strands* coiled around each other to form a double helix, and the DNA fragments in real sequence assembly come from both strands, without exactly knowing the origin of any specific fragment.

1.3 NGS introduction and its consequences

Before next-generation sequencing was introduced the go-to technique was the Sanger method. It's the "original" sequencing technique that allowed for the first time to actually *read* pieces of genetic material. However it soon became evident that the Sanger method was not ideal in order to sequence large sections of DNA. Even if the introduction of shotgun sequencing alongside the Sanger method solved the issue of sequencing larger section of DNA, it was still too slow and expensive. The revolution was the introduction of NGS (Next Generation Sequencing) technologies. The core philosophy of massive parallel sequencing use in NGS is adapted from shotgun sequencing. The read length (actual number of continuous sequenced bases) for NGS is much shorter than that attained by Sanger sequencing.[2] In this aspect Sanger sequencing performs better than NGS but the other advantages achieved by using NGS make up for this disparity. NGS technologies have demonstrated the capacity to sequence DNA at unprecedented speed, thereby enabling previously unimaginable scientific achievements. But the massive data produced by NGS also presents a significant challenge for data storage, analyses, and management solutions. Advanced bioinformatics tools are essential for the successful application of NGS technology. [2]

2 Data structures in Bioinformatics

A relevant aspect of Bioinformatics is the large quantity of data it needs to manage. This prompted since the beginning of the "Bioinformatics era" the need to find efficient ways to manipulate huge datasets. From a programming point of view this is translated in finding efficient data structures that properly represent the biological data. Finding efficient data structures not only reduces the storage/memory usage, which is the primary objective of the algorithm implemented in this paper, but it usually even enhances time performance. For example when a program does a large amount of Input/Output operations, compressing the data enhances both space and time efficiency. In this sense, data structures for compression and indexing are important area of research. Before analyzing the more complex data structures constructed by our algorithm we will introduce two data structures of fundamental importance in computational biology, on which the LCP-array and the BWT are based.

2.1 Suffix tree

A suffix tree is a data structures that exposes the internal structure of a string. Strings are the base components that basically every computational biology algorithm needs to manipulate. In particular the classic application for suffix trees is the *substring problem.*[11] The substring problem consists in finding out if a given string S is contained (i.e. it's a substring), in a longer string T called *text.* Let m and n be the length of T and S respectively. After $\mathcal{O}(m)$, or linear, preprocessing time, one must be able to take in any random string S and in $\mathcal{O}(n)$ time either find an occurrence of S in T or determine that S is not contained in T. This means that the preprocessing time takes time proportional to the length of the text, but after that, the search of S is done in time proportional to the length of S, *independent* of the length of T. These bounds are achieved with the use of a suffix tree. The suffix tree is built in $\mathcal{O}(m)$ time, and exposes important information about the text string in the preprocessing phase. Thereafter, whenever a string of length $\mathcal{O}(n)$ is input, the algorithm searches for it in $\mathcal{O}(n)$ time using that suffix tree.

The concept of suffix tree was first introduced by Weiner [22] in 1973. The construction was, since then, simplified many times. First by McCreight [17] and then by Ukkonen [21]. All these algorithms have a linear running time for fixed alphabets, and a general worst-case running time of $\mathcal{O}(n \log n)$. Farach [8] gave the first suffix tree construction that is optimal for all alpha-

bets. In particular, his solution was the first one to be optimal for alphabets over integers in a polynomial range. We will not examine these more complex linear-time suffix tree construction algorithms, but we will show a naive method that has a quadratic running time.

2.1.1 Basic definitions

Suffix trees are used in a wide range of application, therefore we will not refer to the starting string as T (text) but as the more general and abstract string S. The alphabet is assumed finite and known.

- The *suffix* and *prefix* of length l of a generic m-character string S are the substrings $S[m-l+1:m]$ (denoted by $S[m-l+1:]$) and $S[1:l]$ (denoted by $S[:l]$) respectively. Then the *l-suffix* and *l-prefix* of a string S is the suffix and prefix with length l, respectively [5].
- A suffix tree Z for S is a rooted directed tree with exactly m leaves numbered from 1 to m.
- Each internal node, other than the root, has at least 2 children and each edge is labeled with nonempty substring of S.
- No two edges out of a node can have edge-labels beginning with the same character.
- The string obtained by concatenating all the string-labels found on the path from the root to leaf i spells out the suffix of S that starts at position i. That is, it spells out $S[i...m]$.

This definition of a suffix tree does not guarantee the existence of a suffix tree for any string S. Problems arise when one *suffix* of S matches a *prefix* of another suffix of S. In this case the path of the first suffix would not end at a leaf. This definition guarantees the existence of a suffix tree only for those strings in which the last character does not appear in the rest of the string. So, a solution to the problem is extending the string S with an additional character at the end, called termination symbol, that is not in the same alphabet that S is taken from. The usage of a termination symbol is common practice in computational biology problems and, as usual in bioinformatics literature, we will use the symbol '$' . From now on every string S is assumed to be extended with the termination symbol $, even if not explicitly shown.

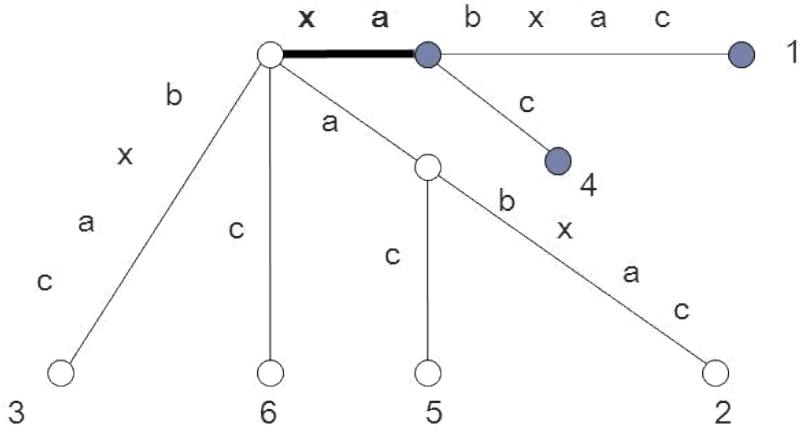

Figure 2: T = $xabxac$, P = xa. We can deduce that the pattern occurs in the text starting at positions 1 and 4

Definitions

- Consider a path from the root that ends at a node. The *label of a path* of that path is the ordered concatenation of the substrings labeling the edges of that path.
- Given a node v, the *string-depth* of v is the length of v's label.

We will know show how the exact matching problem on a text T can be easily solved when possessing the correspondent suffix tree Z.

2.1.2 Suffix trees and exact matching

Given a pattern P of length n and a text T of length m suffix trees can be used to find all occurrences of P in T in $\mathcal{O}(n+m)$ time:

> Build a suffix tree Z for text T in $\mathcal{O}(m)$ time. Then, match the characters of P along the unique path in Z until either P is exhausted or no more matches are possible. In the latter case, P does not appear anywhere in T. In the former case, every leaf in the subtree below the point of the last match is numbered with a starting location of P in T, and every starting location of P in T numbers such a leaf. [11]

The latter case is pretty simple. By traversing the suffix tree down from the root along the path which label matches the pattern, if at some point P is

not exhausted but still there are no matching paths out of the current node, it clearly means that P is nowhere to be found in T since it *doesn't match any prefix of any suffix*. The key to understanding the former case (when at some point P is exhausted) is to note that P occurs in T starting at position j if and only if P occurs as a prefix of the suffix $T[j..m]$. This happens if only if P labels at least an initial part of a path from the root to leaf j. If P is exhausted then P occurs somewhere in T. If P is exhausted at the "end" of the tree, i.e. at a leaf, then P occurs in T only once starting at the position that corresponds to the numeration of the leaf. Otherwise, if P is exhausted at a non-leaf node (see Figure 2) P occurs multiple times in T; precisely P occurs in T a number of times equal to the number leafs in the subtree rooted at the node where P is exhausted. If this is the case, the algorithm can find all the starting positions of P in T by traversing the subtree below the end of the matching path, collecting position numbers written at the leaves. All occurrences of P can therefore be found in $\mathcal{O}(n+m)$ time.

There are further improvements to this method that will not be discussed; instead we will conclude the discussion on suffix trees by showing a simple algorithm for suffix tree construction in quadratic time.

2.1.3 Simple suffix tree construction algorithm

Given a string S of length m this simple method first enters a single edge for suffix $S[1..m]\$$ (the entire string) into the tree. Afterwards it iteratively enters suffix $S[i..m]\$$ into the tree, for i increasing from 2 to m. Let N_i denote the intermediate tree that encodes all the suffixes from 1 to i. In detail N_1 consists of a single edge between the root and a leaf labeled 1. The edge is labeled with the string $S\$$. Tree $N_i + 1$ is constructed from N_i as follows:

> Starting at the root of N_i find the longest path from the root whose label matches a prefix of $S[i+1..m]\$$. This path is found by successively comparing and matching characters in suffix $S[i+1..m]\$$ to characters along a unique path from the root, until no further matches are possible. The matching path is unique because no two edges out of a node can have labels that begin with the same character. At some point, no further matches are possible because no suffix of S\$ is a prefix of any other suffix of S\$. When that point is reached, the algorithm is either at a node, w say, or it is in the middle of an edge. If it is in the middle of an edge, (u, v) say, then it breaks edge (u, v) into two edges by

> inserting a new node, called w, just after the last character on the edge that matched a character in the current suffix($S[i+1..m]$\$). The new edge (u, w) is labeled with the part of the (u, v) label that matched with $S[i+1..m]$\$, and the new edge (w, v) is labeled with the remaining part of the (u, v) label. Then (whether a new node w was created or whether one already existed at the point where the match ended), the algorithm creates a new edge $(w, i+1)$ running from w to a new leaf labeled $i+1$, and it labels the new edge with the unmatched part of suffix $S[i+1..m]$\$. [11].

The tree now contains a unique path from the root to leaf $i+1$, and this path has the label $S[i+1..m]$\$. Assuming a bounded-size alphabet, the above naive method take $\mathcal{O}(m^2)$ time to build a suffix tree for the string S of length m.

We will now introduce suffix arrays, a more space efficient data structure based on suffix trees that has important relations with the LCP-array constructed by the algorithm described in this paper.

2.2 Suffix array and LCP

One of the most important properties that suffix arrays have but suffix trees don't is the **alphabet independence**. Time and space bounds for most strings problems are influenced by the size of the alphabet Σ, the larger the alphabet, the larger the problem. For that reason, sometimes it is explicitly referred to those time and space bounds in terms of alphabet size ($|\Sigma|$). In this case, we would refer to the construction time for suffix tree as $\mathcal{O}(m \log |\Sigma|)$, where m is the size of the string. More completely the Weiner, Ukkonen, and McCreight algorithms that we briefly mentioned all either require $\Theta(m|\Sigma|)$ space, or the $\mathcal{O}(m)$ time bound should be replaced with the minimum of $\mathcal{O}(m \log m)$ and $\mathcal{O}(m \log |\Sigma|)$. Similarly, searching for a pattern P using a suffix tree can be done with $\mathcal{O}(|P|)$ comparisons only if we use $\Theta(m|\Sigma|)$ space, otherwise we must allow the minimum of $\mathcal{O}(|P| \log m)$ and $\mathcal{O}(|P| \log |\Sigma|)$ comparisons during each search. For these reasons, a suffix tree may require too much space to be practical in some applications. In the context of the substring problem, a fixed string T will be searched many times. In this case the most important bounds are:

- Time needed for the search of the pattern in the text.
- Space used by the fixed data structure representing T.

The space used in the construction of the data structure representing T is not critical, although it should be still "reasonable". Usually these two bounds are not completely unrelated, in fact a very space efficient data structure is often the cause of a time-complexity improvement. In 1993 Manber and Myers [16] proposed a new data structure, called *suffix array*, that is very space efficient, yet contains useful information about the internal structure of a string. These informations can be used to efficiently solve many string problems, in particular the exact matching problem, almost as efficiently as with a suffix tree.

Definition

- Given a m-character string T, a **suffix array** for T, called A, is an array of the integers in the range 1 to m, specifying the lexicographic order of the m suffixes of string T.

That is, the suffix starting at position $A(1)$ of T is the lexically smallest suffix. In general suffix at position $A(i)$ is lexically less than suffix at position $A(i+1)$. Even in this contest every string should be assumed extended with the termination symbol \$, which is considered to be lexically *less* than any other character in the alphabet.
Example of a suffix array for $T = banana$:

i	1	2	3	4	5	6	7
$T[i]$	b	a	n	a	n	a	\$

The text has the following suffixes:

$Suffix$	i
banana\$	1
anana\$	2
nana\$	3
ana\$	4
na\$	5
a\$	6
\$	7

The suffixes can be sorted in ascending order:

$Suffix$	i
\$	7
a\$	6
ana\$	4
anana\$	2
banana\$	1
na\$	5
nana\$	3

The **suffix array** A contains the starting positions of these sorted suffixes:

i	1	2	3	4	5	6	7
$A[i]$	7	6	4	2	1	5	3

Notice that the suffix array holds only integers and hence contains no information about the alphabet used in string T. Therefore the space required by suffix arrays is modest: for a string of length m, the array can be stored in exactly m computer words, assuming a word of at least $\log m$ bits.

In combination with the LCP-array, that consists of another $2m$ values, the suffix array can be used to find all occurrences in T of a pattern P in $\mathcal{O}(n + \log_2 m)$ single-character comparison and bookkeeping operations. Moreover, this bound is independent of the alphabet size. Since for most problems of interest $\log_2 m$ is $\mathcal{O}(n)$, the substring problem is solved by using suffix arrays as efficiently as by using suffix trees [11].

Suffix arrays are closely related to suffix trees:

- Suffix arrays can be constructed by performing a depth-first (i.e. linear time) traversal of a suffix tree. The suffix array corresponds to the leaf-labels given in the order in which these are visited during the traversal, if edges are visited in the lexicographical order of their first character. Nevertheless it's possible, and more efficient, to build a suffix array without first constructing a suffix tree and then converting it to a suffix array.
- It has been shown [1] that suffix arrays enhanced with additional information (like the LCP-array) can solve the same problems as suffix trees in the same time complexity.

Definition

> $Lcp[i, j]$ is the length of the longest common prefix of the suffixes specified in positions i and j of A. That is, $Lcp[i, j]$ is the length of the longest prefix common to suffix $A[i]$ and suffix $A[j]$.

The **LCP-array** was first introduced as a "super accelerant" for exact string matching as support to the suffix-array. In this case the LCP-array values are computed in the preprocessing phase. In fact it is possible to compute the LCP during the depth-first tree traversal used to build the suffix array. LCP values are used in the exact matching algorithm to skip as many character comparison as possible; since $Lcp[i, j]$ indicates the number of consecutive initial character common to two strings (suffix $A[i]$ and suffix $A[j]$), it directly indicates how many comparison can be skipped.

The LCP has other important application in computational biology and not always as support to the suffix array. In fact, in certain applications, the space requirements of suffix arrays may *still be prohibitive*. Such requirements motivated the development of even more space-efficient data structures, some of which are based on the BWT, like the FM-index. Some fundamental bioinformatics problems such as genome assembly, require an all-against-all comparison among the input strings, 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 [18]. This fact justifies to search for extremely efficient algorithms to compute the BWT and the LCP array on a collection of strings.

2.3 BWT

The **Burrows-Wheeler Transform** is a *reversible* transformation of a string. It was first introduced in 1994 [6] and its first application was in data compression. The BWT by itself does **not** perform the compression, it simply *permutes* the order of the character of the string. The permutation is done in order to facilitate the string's compression. This happens because the BWT exposes repeated similarities inside the string. For example, if the original string had several substrings that occurred often, then the transformed string will have several places where a single character is repeated multiple times in a row. This is useful for compression, since techniques like move-to-front transform and run-length encoding, can easily exploit repeated sequences of characters inside a string [6]. Furthermore the transformation is **reversible**, without needing to store any additional data. This makes the

BWT a space-free improvement to compression algorithms, and the additional computation required for the BWT construction is a convenient trade-off. One of the most important compression programs that uses the BWT as its core subroutine is bzip2 [24].

The actual transformation is done as follows:

- Generation of all possible rotations of the text and creation of a matrix in which every rotation is a row.
- Sorting the rows (i.e. rotations) by lexicographic order.
- The BWT is the last column of the resulting matrix.

Example for $T = BANANA\$$, where \$, as usual, indicates end of the string. In this example as well, we follow the convention that the symbol \$ is lexically less of any character of the alphabet.

Input	All rotations	Lexical sorting	Last column	Output
	BANANA\$	\$BANANA	\$BANAN**A**	
	\$BANANA	A\$BANAN	A\$BANA**N**	
	A\$BANAN	ANA\$BAN	ANA\$BA**N**	
BANANA\$	NA\$BANA	ANANA\$B	ANANA\$**B**	**ANNB\$AA**
	ANA\$BAN	BANANA\$	BANANA**\$**	
	NANA\$BA	NA\$BANA	NA\$BAN**A**	
	ANANA\$B	NANA\$BA	NANA\$B**A**	

The BWT of a string T 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 lexicographic 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 [9]. It is important to note that the algorithm described constructs the BWT of a *collection* of strings, which is a generalization of the procedure described above. One of the advantage of the new theoretical approach discussed in [5] is that it facilitates the extension to a collection of variable-length strings . This possible further improvement is briefly discussed in the final section of this paper, but it is important to note that the implementation is based on the less general case of same-length string collection.

3 Algorithm description

Here we report the algorithm formalization as introduced in [5]. The algorithm computes the Burrows-Wheeler Transform and the Longest Common Prefix array for a collection of string in external memory. Among the first efficient algorithms for this task there are BCRext[3] and extLCP[7] for BWT and LCP construction respectively. The method described here aims to be an efficient alternative to those approaches.

3.1 Basic concepts

Let $\Sigma = \{c_0, c_1, ..., c_\sigma\}$ be a finite alphabet where $c_0 = \$$ (termination symbol), and $c_0 < c_1 < ... < c_\sigma$ where $<$ specifies the lexicographic ordering over alphabet Σ. We consider a collection $S = \{s_1, s_2, ..., s_m\}$ of m strings, where each string s_j consists of k symbols over the alphabet $\Sigma \setminus \{\$\}$ and is terminated by the symbol \$. As already discussed, k is assumed fixed for every string in the collection.

3.1.1 Auxiliary definitions

- $s_j[i]$ denotes the *i-th* symbol of string s_j.
- $s_j[i : t]$ denotes the substring containing the consecutive characters from position i to t of S_j.
- Let V be a generic vector, we denote with $V[1 : q]$ the first q elements of V and with $rank_V(q, x)$ the number of elements equals to x in $V[1 : q]$.
- Let X be the lexicographic ordering of the suffixes of the collection S. The *Suffix Array* A is the $m(k + 1)$-long array where $A[i] = (p, j)$ if and only if the i-th element of X is the p-suffix of string s_j.
- The *Burrows-Wheeler Transform* of S is the $m(k + 1)$-long array B where if $A[i] = (p, j)$ then $B[i]$ is the first symbol of the $(p + 1)$-suffix of s_j if $p < k$, otherwise $B[i] = \$$. In other terms 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 array* of S is the $m(k + 1)$-long array LCP such that $LCP[i] =$ length of the longest prefix shared by suffixes $X[i - 1]$ and $X[i]$. Conventionally, $LCP[1] = -1$.

3.2 BWT construction: Interleave

The construction of the BWT extensively uses the notion of *interleave*. In particular the algorithm computes the BWT starting from two elements: The **partial**-BWT and the encoding I_W. The partial-BWT is a collection of $k+1$ arrays of characters and the *interleave-encoding* I_W is an array of $m(k+1)$ integers. In simple terms, the **interleave** of the $k+1$ arrays is the result of merging those arrays "mediated" by the interleave encoding. In this algorithm the BWT corresponds to the interleave of the partial-BWT and the encoding I_W.

More formally, the interleave W of a generic set of arrays $V_0, V_1, ..., V_n$ 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 the I_W array the encode the information 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 $j = rank_{I_W}(q, I_W[q])$, it is easy to show how to reconstruct W from I_W (see Algorithm 1 where the array $pos[i]$ at line 6 is equal to $rank_{I_W}(q, i)$).

Figure 3 shows an example of an interleave of four arrays V_0, V_1, V_2, V_3 and its encoding.

V_0	V_1	V_2	V_3
T	C	A	A
T	G	C	A
A	G	C	T

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

Figure 3: Example of an interleave W of four arrays V_0, V_1, V_2, V_3.

Algorithm 1: Reconstruct the interleave W from the encoding I_W

```
1 for i ← 0 to n do
2 |   pos[i] ← 0;
3 for q ← 1 to |I_W| do
4 |   i ← I_W[q];
5 |   pos[i] ← pos[i] + 1;
6 |   W[q] ← V_i[pos[i]];
```

3.3 Sketch of the algorithm

Let X_l and B_l, with $0 \leq l \leq k$, be the arrays of length m such that $X_l[i]$ is the $i\text{-}th$ smallest l-suffix of S and $B_l[i]$ is the symbol preceding $X_l[i]$. In particular X_0 and B_0 respectively list the 0-suffixes and the last characters of the input strings in their order in the set S. It follows that B_l is a subsequence of the BWT B of S. Furthermore B is an interleave of the $k+1$ arrays $B_0, B_1, ..., B_k$. Similarly, the lexicographic ordering X of all suffixes of S is an interleave of the arrays $X_0, X_1, ..., X_k$. Let I_B be the encoding of the interleave of arrays $B_0, B_1, ..., B_k$, giving the BWT B, and let I_X be the encoding of the interleave of arrays $X_0, X_1, ..., X_k$, giving X. Then it's possible to show that $I_B = I_X$ [5]. The algorithm exploits this property to compute the BWT as an interleave of I_X, instead of using I_B. Moreover I_X can be simultaneously used to compute the LCP array as well.

The algorithm, differently from BCRext and extLCP, consists of two different phases: in the first phase the partial BWTs $B_0, B_1, ..., B_k$ are computed (see section Section 3.4), while the second phase (Section 3.5) is based on the Holt-McMillan approach [12] and determines I_X, thus allowing to reconstruct B as an interleave of $B_0, B_1, ..., B_k$. In particular the method consists in an external memory radix-sort approach realized by using lists, which can be implemented in files. In order for the implementation to be efficient in contexts were Hard Drives are used (in contrast to Solid-State Drives) every file (list) is accessed sequentially.

3.4 Partial BWT computation

First the input strings $s_1, ..., s_m$ are preprocessed in order to compute $k+1$ arrays $T_0, T_1, ..., T_k$ with length m.

- T_l lists the characters in position $k-l$ of the input strings, such that $T_l[i] = s_i[k-l]$ when $0 \leq l \leq k-1$ and $T_l[i] = \$$ when $l = k$.

It follows that $T_l[i]$ is the symbol preceding the l-suffix of s_i. Clearly, those arrays can be computed in $\mathcal{O}(km)$ time and $\mathcal{O}(km \log \sigma)$ I/O complexity by sequentially reading the input strings. Then Algorithm 2 computes the partial BWTs $B_0, B_1, ..., B_k$ by receiving in input the arrays $T_0, T_1, ..., T_k - 1$ and by using $k+1$ arrays N_l with length m $(0 \leq l \leq k)$, defined as follows:

- $N_l[i] = q$ if and only if the *i-th* element of X_l is the l-suffix of the input strings s_q.

Note that $B_l[i] = s_q[k-l]$ and when $l = k$, $B_k[i] = \$$. In particular, N_0 is the sequence of indexes $\langle 1, 2, 3, ..., |S| \rangle$ and B_0 is the sequence $\langle s_1[k], s_2[k], ..., s_m[k] \rangle$ of the last symbols of the input strings (i.e. the symbols before the sentinels); that is $B_0 = T_0$. So, the algorithm starts with B_0 and N_0. At each iteration l, B_l is computed from N_l which in turn is computed with the support of the set of lists, $\mathcal{P}\ (c_h) = \{\mathcal{P}(c_0),\ \mathcal{P}(c_1),\ ...,\ \mathcal{P}(c_\sigma)\}$ where $c_0, c_1, ..., c_\sigma$ is the lexicographic order of symbols in Σ (i.e. $0 \leq h \leq |\Sigma|$). N_l will be the result of the *ordered concatenation* of each lists in $\mathcal{P}\ (c_h)$. This set is created as follows:

- We perform a linear scan of B_{l-1}. Let i be the index of this scan and let $c_h = B_{l-1}[i]$.
- We append the integer $N_{l-1}[i]$ to $\mathcal{P}\ (c_h)$, where $c_h = B_{l-1}[i]$.
- N_l is then obtained as the concatenation of $\mathcal{P}(c_0)$, $\mathcal{P}(c_1)$, ..., $\mathcal{P}(c_\sigma)$.
- Finally we compute B_l by performing another linear scan but this time of the freshly calculated N_l. Let $q = N_l[i]$, then $B_l[i] = T_l[q]$.

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 random access, for each l (loop at line 4) the array T_l is kept in RAM, but in different iterations of the loop, with a total space cost of $O(m \log \sigma)$.

Algorithm 2 describes this procedure in detail.

Algorithm 2: Compute the partial BWTs $B_0, B_1, \cdots, B_k$

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

for $i \leftarrow 1$ *to* m **do**
  $B_0[i] \leftarrow T_0[i]$;
  $N_0[i] \leftarrow i$;
for $l \leftarrow 1$ *to* k **do**
  for $h \leftarrow 0$ *to* σ **do**
    $\mathcal{P}(c_h) \leftarrow$ empty list;
  for $i \leftarrow 1$ *to* m **do**
    $c_h \leftarrow B_{l-1}[i]$;
    $q \leftarrow N_{l-1}[i]$;
    Append q to $\mathcal{P}(c_h)$;
  $N_l \leftarrow \mathcal{P}(c_0)\mathcal{P}(c_1)\cdots\mathcal{P}(c_\sigma)$;
  for $i \leftarrow 1$ *to* m **do**
    $q \leftarrow N_l[i]$;
    $B_l[i] \leftarrow T_l[q]$;

3.5 From the partial BWT to the I_X encoding and the LCP array

In this section we will describe the second step of the algorithm that computes the BWT B and the LCP array. In order to compute the BWT B the only element needed at this point of the algorithm is the encoding I_X, since phase 1 already computed the partial BWTs $B_0, B_1, ..., B_k$. In this paper we will not provide a formal proof of correctness for the algorithm; this can be found in [5]. The main idea in this phase 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 their lexicographic order. Similarly to phase one, the I_X encoding and the LCP array will be computed iteratively. Each iteration p computes $I_{X_{p+1}}$ and LCP_{p+1} from I_{X_p} and LCP_p respectively. Similarly to the partial BWT computation, even $I_{X_{p+1}}$ and LCP_{p+1} are constructed as the ordered concatenation of $\sigma + 1$ additional lists generated at every iteration from the partial BWT and I_{X_p} or LCP_p for BWT and LCP respectively. They key problem now becomes how are these $\sigma + 1$ lists created; we will separately show their construction.

Algorithm 3: Compute $I_{X^{p+1}}$ and LCP_{p+1} from I_{X^p} and LCP_p

$\mathcal{I}(c_0) \leftarrow 0, 0, \cdots, 0$;
$\mathcal{I}(c_1), \ldots, \mathcal{I}(c_\sigma) \leftarrow$ empty lists;
$\mathcal{L}(c_0) \leftarrow -1, 0, \cdots, 0$;
$\mathcal{L}(c_1), \ldots, \mathcal{L}(c_\sigma) \leftarrow$ empty lists;
foreach $c \in \{c_1, \cdots, c_\sigma\}$ **do**
  $\mathcal{L}(c) \leftarrow$ the list $< 0 >$;
$\alpha \leftarrow \{-1, -1, \ldots, -1\}$;
for $i \leftarrow 1$ *to* $|I_{X^p}|$ **do**
  $l \leftarrow I_{X^p}[i]$;
  $c \leftarrow B_l[rank_{I_{X^p}}(i, l)]$; `// is the character preceding the current suffix`
  if $c \neq \$$ **then**
    Append $(l+1)$ to $\mathcal{I}(c)$;
  $lcp \leftarrow LCP_p[i]$;
  foreach $d \in \{c_1 \cdots, c_\sigma\}$ **do**
    $\alpha[d] \leftarrow \min\{\alpha[d], lcp\}$;
  if $c \neq \$$ *and* $\alpha[c] \geq 0$ **then**
    Append $\alpha[c] + 1$ to $\mathcal{L}(c)$;
  $\alpha[c] = \infty$;
$I_{X^{p+1}} \leftarrow \mathcal{I}(c_0)\mathcal{I}(c_1)\cdots\mathcal{I}(c_\sigma)$;
$LCP_{p+1} \leftarrow \mathcal{L}(c_0)\mathcal{L}(c_1)\cdots\mathcal{L}(c_\sigma)$;

3.5.1 Lists for I_{X_p} construction

To build $I_{X^{p+1}}$ from I_{X^p}, the algorithm builds a set of $\sigma + 1$ lists $\mathcal{I}(c_0), \mathcal{I}(c_1), \cdots, \mathcal{I}(c_\sigma)$ 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 over the iterations and is always composed of m 0s. Finally, the algorithm produces $I_{X^{p+1}}$ as the concatenation $\mathcal{I}(c_0)\mathcal{I}(c_1)\cdots\mathcal{I}(c_\sigma)$. The rest of the lists (with $\sigma > 0$) is created as follows:

- Linear scan of I_{X_p}: for each position i, $l = I_{X_p}[i]$ and $c = B_l[rank_{I_{X_p}}(i, l)]$ (i.e. the character preceding the current suffix). 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.

3.5.2 Lists for LCP_p construction

To build LCP_{p+1} from LCP_p, the algorithm builds a set of $\sigma + 1$ lists $\mathcal{L}(c_0), \mathcal{L}(c_1), \cdots, \mathcal{L}(c_\sigma)$ that is the partitioning of the elements of LCP_{p+1} by the first character c_i $(0 \leq i \leq \sigma)$ of their related suffixes. Since the list $\mathcal{L}(c_0 = \$)$ is related to empty suffixes, it is fixed over the iterations and is composed of -1 followed by $m - 1$ 0s. Moreover, observe that the first element of the list $\mathcal{L}(c_i)$ $(1 \leq i \leq \sigma)$ is always 0. Finally, Algorithm 3 concatenates all the lists $\mathcal{L}(c_0), \mathcal{L}(c_1), \cdots, \mathcal{L}(c_\sigma)$, thus producing LCP_{p+1} (see line 20).

Before giving some more detail, we introduce the function α. Given a position i and a symbol $c \neq \$$, the function $\alpha(i, c)$ is the length of the longest prefix shared by the prefixes $X_p[i]$ and $X_h[i]$, 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 function $\alpha_p(i, c)$ is maintained in the array α of size $\sigma - 1$ initially set to $(\sigma - 1)$ -1s values (see line 7), and updated in the cycle at line 14. Observe that as the index for this array we use a character of the alphabet that must be considered as the number corresponding to the character's position in the lexicographic order (e.g $\$ = 0 \ldots$). Function α (encoded as an array) is used in Algorithm 3 to compute the $\sigma - 1$ lists remaining in order to obtain LCP_{p+1} as follows:

- Linear scan of LCP_p (since LCP_p and I_{X_p} have the same length their linear is done simultaneously): for each position i, $lcp = LCP_p[i]$.
- Then we modify the values in α depending on the value of lcp: if $lcp < \alpha[h]$ $(0 < h \leq \sigma)$ then the new value of $\alpha[h]$ is lcp, else it does not change.
- Finally we append the number $\alpha[c]+1$ (remember that $c = B_l[rank_{I_{X_p}}(i, l)]$ is the character preceding the current suffix) to the list $\mathcal{L}(c)$ **if** $c \neq \$$ and $\alpha[c] \geq 0$.

Algorithm 3 showed how to compute $I_{X_{p+1}}$ from I_{X_p} and LCP_{p+1} from LCP_p. LCP_0 and I_{X_0} are fixed, since their meaning is strictly related to basic information (like the length of the longest common prefix between to strings of length 0). Algorithm 4 iterates Algorithm 3 to finally compute the LCP array and the BWT encoding. The iteration stops when the maximum value $\max_i\{LCP_p[i]\}$ in the array LCP_p is less than p. In fact, this 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.

Algorithm 4: 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

Compute $T_0, \ldots, T_k$ from $s_1, \ldots, s_m$;

Apply Algorithm 2 to compute $B_0, \ldots B_k$ from $T_0, \ldots, T_k$;

$I_{X^0} \leftarrow m$ 0s, m 1s, ..., m ks;

$LCP_0 \leftarrow -1, 0, 0, \ldots, 0$;

$p \leftarrow 0$;

while $\max_i\{LCP_{p+1}[i]\} = p + 1$ **do**

  Apply Algorithm 3 to compute $I_{X^{p+1}}$ and LCP_{p+1} from I_{X^p}, LCP_p and the lists $B_0, \ldots B_k$;

  $p \leftarrow p + 1$;

Reconstruct B from I_{X^p} and $B_0, \cdots, B_k$;

Output: (B, LCP_p)

4 Algorithm implementation

The implementation of the algorithm discussed in Section 3 is described here. Given the importance of memory and Input/Output (I/O) management, the *C* programming language has been the choice for this project, since it offers total control over main memory and generally elevate time and space efficiency. Although when first introduced (1972) *C* was considered a high-level language (in contrast to the widely used *assembly language*), nowadays it's the go-to choice when performance and low-level access to memory are key. This comes at a cost: compared to more recent programming languages, programming in *C* is generally more difficult, mostly because it implicitly forces the programmer to deeply understand the low-level architecture of a computer, in order to take full control of it. Furthermore it does not provide nearly as many built-in functionalities as other modern high-level languages, making the programming process more tedious, but in some ways more useful from a learning perspective and more rewarding.

The choice of the programming language somewhat influenced the choice of GNU/Linux as the operative system used to develop this program. First of all, the origin of *C* has been closely associated with the Unix operative system for which it was developed, since the system and most of its programs are written in *C*. For this reason all Unix-like operating systems (like GNU/Linux) provide exceptional support to developers for this language. For example, the majority of Linux distributions have pre-installed a *C* compiler and a text-editor, which are the indispensable tools for developing in *C*. In the process of writing this program many other tools have been used such as git, valgrind, bash, python and many others. Git is a powerful free and open source distributed version control system that, among many other things, efficiently allows to track every change to files of the project. The code of this program is available on GitHub (`https://github.com/AlgoLab/bwt-lcp-em`), a web interface of git and hosting service.

4.1 FASTA file format

The input of the program is a collection of strings. The only file format supported is the **FASTA format**: a simple text-based format for representing nucleotide sequences or peptide sequences, in which nucleotides or amino acids are represented using single-letter codes. The program limits the alphabet Σ from which the codes are taken to $\{A, C, G, T, \}$ and therefore does not actually work with every FASTA file; the FASTA standard allows larger alphabets that can contain gaps or alignment characters, and characters that represent more than one nucleotide (for example S = Cytosine or Guanine). Expanding the alphabet may be a possible improvement to the program but it's not a simple task at all, mostly because of how the alphabet is encoded (Section 4.2.1). The FASTA standard allows a "description line" to be inserted before every sequence that must begin with either the character '>' or ';'. For testing purposes a simple python script that generates FASTA files with random strings from the alphabet Σ has been written; here is an example of output with 10 random strings of length 10:

```
>seq0
TTTTCCCCGT
>seq1
TGGTTCGGAT
>seq2
GATCTTAGCA
>seq3
GTCAAAGAGG
>seq4
CCTTCCCAGA
>seq5
CGCGTCCGAA
>seq6
CAAGCTTAGG
>seq7
GGTGGTGCTG
>seq8
TGATCTGAGG
>seq9
CCCCCATACG
```

4.1.1 Klib: parsing input files

Parsing formats, even as simple as FASTA, can be a laborious task in *C*; therefore this part of the program is delegated to the highly efficient external library **Klib** (https://github.com/attractivechaos/klib). Klib is a standalone and lightweight *C* library, in which most components are independent of external libraries (except the standard *C* library), and independent of each other. Some components of this library, such as hash table, B-tree, vector and sorting algorithms, are among the most efficient implementations of similar algorithms or data structures in all programming languages, in terms of both speed and memory use [25]. Using this library allows the program to extend its support to FASTA files optionally compressed with the **zlib** [26] compression program.

4.2 Memory management

In every step of the programming phase the top priority was to find efficient ways to keep the RAM usage as low as possible. Most of the work was already done in designing the algorithm in such a way that intensively uses external memory whenever possible. The part of the algorithm that requires the most amount of memory is the final part of the partial BWT computation where, due to random access, we must keep in RAM a file the size of which linearly depends on the amount of input strings (variable m in Section 3). Keep in mind the the two variable m and k which respectively represent the number of input strings and the length of each string highly differ from each other. As already discussed in the introduction the length of the DNA reads (k) in NGS technologies do not exceed the 400-500 bases. On the other hand the human genome consists of roughly 3,2 billions nucleotides and since to exploit overlap between reads there must be sufficient coverage (i.e. many reads that contains the same character on the original string) in big projects the value of m can reach the order of billions.

The other part of the program that could seem to use a large amount of RAM is line 10 of algorithm 3, where we must access k files at the same time at the position $B_l[rank_{I_{X^p}}(i, l)]$ (with $0 \leq l \leq k$). In reality the rank function in this case simply means that we are accessing this files *sequentially* and therefore there is no need to store them in RAM. Nevertheless this observation brings up another important point of discussion about memory consumption: the fundamental role of I/O buffers on the performance of the program.

4.2.1 I/O buffering and the value of compression

Most of the execution time of this program is spent in I/O operations. This is a consequence of the intensive usage of external memory and its management is probably the most important parameter that influences the performance of the program. Storing data in external memory and sequentially read them may seem an operation that does not require the usage of internal memory at all. While this is theoretically possible, in practice it never is because of the huge drop in time performance it would cause. Trading off time performance in exchange of less RAM consumption is a known aspect of programming. Nonetheless executing I/O operations on an *unbuffered* stream is usually a not convenient compromise: the amount of space saved is almost never worth the large drop in time performance. For this reason usually every I/O stream is by default *buffered*, which means that every stream is associated with an array (its *buffer*) that stores values (in main memory) and performs I/O operations only when the buffer is full, drastically reducing the amount of external memory access, which is usually one of the slowest components of a computer program.

In this context any type of technique that reduces I/O operations is a valuable tool. In this program it was possible through exploiting low-level bitwise operations to halve the size of files in external memory, and therefore reducing I/O complexity. First of all we must consider that the smallest addressable block of memory is called Byte and corresponds to 8 bits. As a consequence the lower limit to the size of a variable in memory corresponds to 8 bits. The alphabet of characters used in the program is $\Sigma = \{A, C, G, T, \} \cup \{\$, @\}$. The \$ is the termination symbol and '@' is a character that needed to be used as consequence of the compression. Since the variable representing a character taken from this alphabet can only assume 6 different values the lowest amount of bits that such a variable can use is $\lceil \log_2 6 \rceil = 3$. The problem is that this value is lower than 8, the lowest the size of a variable can get. Solving this problem requires to encode the information of more than one character in a single variable. Since using 4 bits (instead of the more space efficient 3 bits) per character (exactly 2 characters encoded in 1 variable) simplifies the compression task, during the implementation of the program the decision of using a 4-bit encoding was taken. Nonetheless this makes every other step of the program more complicated since before and after every I/O operation the program has to encode/decode every character. Character '@' was introduced to handle the cases where the number of reads (m) is an odd number. In fact, this would cause some variable to only have the first 4 bits as "actual" information; in such cases the character '@' is used

to encode this "lack" of information in the second half of the variable.

5 Conclusive experimentation and possible improvements

This last section is devoted to describe the effective performance of the program compared to similar software and to explore aspects that can be improved.

5.1 Comparison with similar software

The input files ranged from 1 million to 1 billion reads. The other program used were BEETL, that implements the extLCP algorithm, and gsa-is that does not utilize external memory but its time performance can still be a useful parameter to show differences between external and main memory algorithms. Gsa-is was considerably faster compared to both BEETL and bwt-lcp-em (the program discussed in this paper) but soon enough it ended up using too much main memory and crashed. Bwt-lcp-em uses an extremely low amount of main memory, on average a tenth compared to BEETL, but is considerably slower as well (7 times slower on average). Overall these results are a good starting point, considering how much room of improvement there is. The first aspect to consider is the compromise between memory usage and time performance.

5.1.1 A better compromise between RAM usage and time performance

At the time this paper is written there are lots of parts in the code where small workarounds would enhance time performance but there is only few that would have a huge impact. As already stated, most of the execution time the program does I/O operations and finding ways to improve this aspect will result in the highest time performance gain. The solution lies in finding the best size for the I/O buffers. This is not at all a simple task, since it requires a deep knowledge of how I/O works at low level and depends on many variables: type of external memory drive, operating system and so on. For example the GNU C library documentation [27], states that the default value of the size of the stream buffers is determined in such a way to make

I/O efficient on the particular system in usage. The default value is a good choice in normal circumstances, but considering the key aspect of I/O in this particular case, finding the best way to increase the buffer size will probably lead to overall performance improvement.

5.2 Enhancing compression

Another key area of improvement is enhancing the compression technique. This area and the size of the I/O buffers are the two most influencing parameters of overall performance. The first idea that comes to mind is switching from a 4-bit encoding to a 3-bit encoding. This complicates the implementation by quite a lot because instead of simply having 2 character encoded in 1 byte the distribution would be like this:

- The first byte would contain two characters plus two bits of a third character $(3 + 3 + 2 = 8)$
- The second byte would contain the leftover 1 bit from the previous byte plus two full characters (6 bits) and 1 bit of a fourth character $(1 + 3 + 3 + 1 = 8)$
- The third byte would "end" the chain by containing the 2 bits remaining from the previous byte plus two full characters. $(2 + 3 + 3 = 8)$

Although this may seem the best compression possible, given that it's the least amount of bits to encode the alphabet, there are more advanced techniques that can improve the compression even further. For example the usage of Huffman coding [13] can be extremely useful in this particular context. The fundamental idea is to encode characters that presents more frequently with shorter codes, and those that are no so frequent with longer ones. Considering that the strings processed by this algorithm are not random but consists of DNA-fragments, the usage of Huffman coding after an analysis of the frequency of each nucleotide in the DNA is almost certainly one of the best option to improve the compression rate.

5.3 Variable length reads extension

At last we will discuss of the extension on the program in order to support collection of strings with variable length. We already mentioned that a point of strength of the algorithm's design is the theoretically easy extension to this case, consequence of the many linear scans performed. Nonetheless such extension is not naive. During the implementation of the program an attempt to extend the program to this more general case was made. The idea was to use a special character, say #, to add to each string that was shorter than the longest in order to re-enter the case where all the strings had the same length. The problem of this approach is that there is no way to insert the character # in the lexicographic order of symbols in order to not influence the final ordering in the Burrows-Wheeler Transform. The solution of this problem lies in correctly exploiting linear scans of files to make them independent from their length.

References

[1] M. I. Abouelhoda, S. Kurz, E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2004.

[2] A. Badr, J. Zhang, R. Chiodini, G. Zhang. The impact of next generation sequencing on genomics, 2011.

[3] M. Bauer, A. Cox, and G. Rosone. Lightweight BWT construction for very large string collections. In *Combinatorial Pattern Matching*, Springer, 2011.

[4] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis. Linear approximation of shortest superstrings, 1994.

[5] P. Bonizzoni, G. Della Vedova, Y. Pirola, M. Previtali and R. Rizzi. Computing the BWT and LCP array of a set of strings in external memory, 2017.

[6] M. Burrows and D. J. Wheeler, A block sorting lossless data compression algorithm. Technical report, Digital Systems Research Center, 1994.

[7] A. Cox, F. Garofalo, G. Rosone, and M. Sciortino. Lightweight LCP construction for very large collections of strings. *Journal of Computational Biology*, 2016.

[8] M. Farach, Optimal suffix tree construction with large alphabets. *38th IEEE Symposium on foundations of computer science.*, 1997.

[9] P. Ferragina, G. Manzini. Indexing compressed text. *J. of the ACM*, 2005.

[10] M. Garey and D. Johnson. *Computers and Intractability.* Freeman, San Francisco, 1979

[11] D. Gusfield, *Algorithms on strings, trees, and sequences : computer science and computational biology*, 1997.

[12] J. Holt and L. McMillan. Merging of multi-string BWTs with applications.*Bioinformatics*, 2014.

[13] D: Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 1952.

[14] Haim Kaplan, Moshe Lewenstein, Nira Shafrir, and Maxim Sviridenko. Approximation algorithms for asymmetric tsp by decomposing directed regular multigraphs, 2005.

[15] E. Lander and M. Waterman. Genomic mapping by fingerprinting random clones: a mathematical analysis. *Genomics*, 1988.

[16] U. Manber and G. Myers. Suffix arrays: a new method for on-line search. *SIAM J. Comput.*, 1993.

[17] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 1976.

[18] E. Myers. The fragment assembly string graph. *Bioinformatics*, 2005.

[19] M. V. Olson. A time to sequence. *Science*, 1995.

[20] E. S. Sweedyk. *A 2 1/2 approximation algorithm for shortest common superstring.* PhD thesis, Univ. Calif., Berkeley, Dept. Computer Science, 1995.

[21] E. Ukkonen, On-line construction of suffix-trees. *Algorithmica*, 1995.

[22] P. Weiner. Linear pattern matching algorithms. *Proc. of the 14th IEEE Symp. on Switching and Automata Theory.*, 1973.

[23] `https://en.wikipedia.org/wiki/Sequence_assembly`, Introduction.

[24] `http://www.bzip.org/`.

[25] `http://attractivechaos.github.io/klib/#About`, Klib library documentation.

[26] `https://zlib.net/`, Zlib compression program offical website.

[27] `https://www.gnu.org/software/libc/manual/`, GNU C library documentation.