The colored longest common prefix array computed via sequential scans[⋆]

Fabio Garofalo[1], Giovanna Rosone[2⋆⋆],
Marinella Sciortino[1], and Davide Verzotto[2]

[1] University of Palermo, Palermo, Italy
garofalo_uni@yahoo.it, marinella.sciortino@unipa.it
[2] University of Pisa, Pisa, Italy
giovanna.rosone@unipi.it, davide.verzotto@di.unipi.it

Abstract. Due to the increased availability of large datasets of biological sequences, the tools for sequence comparison are now relying on efficient alignment-free approaches to a greater extent. Most of the alignment-free approaches require the computation of statistics of the sequences in the dataset. Such computations become impractical in internal memory when very large collections of long sequences are considered. In this paper, we present a new conceptual data structure, the *colored longest common prefix array* (cLCP), that allows to efficiently tackle several problems with an alignment-free approach. In fact, we show that such a data structure can be computed via sequential scans in semi-external memory. By using cLCP, we propose an efficient lightweight strategy to solve the *multi-string Average Common Substring (*ACS*) problem*, that consists in the pairwise comparison of a single string against a collection of m strings simultaneously, in order to obtain m ACS induced distances. Experimental results confirm the effectiveness of our approach.

Keywords: Longest common prefix · Average common substring · Matching statistics · Burrows-Wheeler transform · Alignment-free methods.

1 Introduction

The rapid increase in the availability of large sets of biological sequences observed in the last two decades, particularly triggered by the human sequencing project, posed several challenges in the analysis of such data. So far, traditional methods based on sequence alignment worked well for small and closely related sequences, but scaling these approaches up to multiple divergent sequences, especially of large genomes and proteomes, is a difficult task. To keep pace with this, several algorithms that go beyond the concept of sequence alignment have been developed,

[⋆] G.R. and M.S. are partially and D.V. is supported by the project Italian MIUR-SIR CMACBioSeq ("Combinatorial methods for analysis and compression of biological sequences") grant n. RBSI146R5L.

[⋆⋆] Corresponding author

called alignment-free [35]. Alignment-free approaches have been explored in several large-scale biological applications ranging, for instance, from DNA sequence comparison [12, 28, 14, 19, 27] to whole-genome phylogeny construction [34, 15, 13, 23, 33] and the classification of protein sequences [14]. Most alignment-free approaches above mentioned require, each with its own specific approach and with the use of appropriate data structures, the computation of statistics of the sequences of the analyzed collections. However, it is interesting to note that the increasing number of completely sequenced genomes has caused the computation of many statistics to be impracticable in internal memory, determining the need for lightweight strategies for the comparative analysis of very large collections of long sequences.

In this paper, we propose a new conceptual data structure, the *colored longest common prefix array* (cLCP), that implicitly stores all the information necessary for computation of statistics on distinguishing, repeating or matching strings within a collection or different collections. Loosely speaking, given a collection $\mathcal{S}$, in which each string (or subset of strings) can be identified by a specific color, we can generally define cLCP as an integer array representing the longest common prefix between any specific suffix of a string $s_r \in \mathcal{S}$ and the nearest suffix of a specific string $s_t \in \mathcal{S}$ in the sorted list of suffixes of $\mathcal{S}$. Here, we assume that $\mathcal{S}$ is partitioned in two subsets and consider the suffixes of strings belonging to different subsets, but we remark that one can consider any situation and note also that the definition can be easily adapted to more than two sets. We also show that cLCP can be computed via sequential scans and therefore acquires the characteristics of an appropriate structure for analyzing large collections of strings stored in external memory.

Such a data structure can be used in several applicative contexts. In this paper we explore the multi-string Average Common Substring (ACS) [34] problem. More specifically, the ACS measure is a simple and effective alignment-free method for pairwise string comparison [13, 33], based on the concept of *matching statistics* (MS) [12, 22, 28]. Given two strings s and t, it can be defined as a couple of arrays $\mathsf{MS}(s,t)$ and $\mathsf{MS}(t,s)$ that store, at each position i, the length of the longest substring that starts at position i of the string given as first parameter that is also a substring of the string given as second parameter.

ACS approach has been employed in several biological applications [34, 16, 23, 15, 17]. Generalization of measures based on longest matches with mismatches have been proposed in [7], also with distributed approaches [21]. Similarly to [31], we define the *multi-string* ACS *problem* as the pairwise comparison of a single string, say $s_\chi \in \mathcal{S}^0$ of length n_χ, against a set of m strings, say $s_r \in \mathcal{S}^1$ with $1 \leq r \leq m$, by considering the strings in $\mathcal{S}^1$ all together, in order to obtain m ACS induced distances at once. A major bottleneck in the computation (and application) of ACS and MS initially consisted in the construction of a suffix tree. More recent approaches use efficient indexing structures [9], CDAWG [10], backward search [28] or enhanced suffix arrays [23]. However, to our knowledge, the above mentioned approaches would require a great effort, especially in terms of RAM space, if applied to very large collections of long strings.

In this paper we use cLCP to efficiently solve the above mentioned multi-string ACS problem. Preliminary experimental results show that our algorithm is a competitive tool for the lightweight simultaneous computation of a pairwise distance of a string versus each string in a collection, allowing us to suppose that this data structure and its computational strategy can be used for more general versions of the multi-string ACS problem.

2 Preliminaries

Let $\Sigma = \{c_1, c_2, \ldots, c_\sigma\}$ be a finite ordered alphabet with $c_1 < c_2 < \ldots < c_\sigma$, where $<$ denotes the standard lexicographic order. We consider finite strings such as $s \in \Sigma^*$, where $s[1], s[2], \ldots, s[n]$ denote its characters and $|s| = n$ its length. A *substring* of a string s is written as $s[i, j] = s[i] \cdots s[j]$, with a substring $s[1, j]$ being called a *prefix*, while a substring $s[i, n]$ is referred to as a *suffix*. A range is delimited by a square bracket if the correspondent endpoint is included, whereas a parenthesis means that the endpoint is excluded.

The BWT [11] is a well known reversible string transformation widely used in data compression. The BWT can be extended to a collection of strings $\mathcal{S} = \{s_1, s_2, \ldots, s_m\}$. Such an extension, known as EBWT or multi-string BWT, is a reversible transformation that produces a string (denoted by $\mathsf{ebwt}(\mathcal{S})$) that is a permutation of the characters of all strings in $\mathcal{S}$ [26]. Lightweight implementations of EBWT have been proposed [8, 24, 20]. We append to each string s_i of length n_i a distinct end-marker symbol $\$_i < c_1$ (for implementation purposes, we could simply use a unique end-marker \$ for all strings in $\mathcal{S}$). The output string $\mathsf{ebwt}(\mathcal{S})$ is the concatenation of the symbols (circularly) preceding each suffix of the strings in $\mathcal{S}$, sorted according to the lexicographic order. More in detail, the length of $\mathsf{ebwt}(\mathcal{S})$ is denoted by $N = \sum_{i=1}^{m} n_i + m$ and $\mathsf{ebwt}(\mathcal{S})[i] = x$, with $1 \leq i \leq N$, if x circularly precedes the i-th suffix $s_j[k, n_j + 1]$ (for some $1 \leq j \leq m$ and $1 \leq k \leq n_j + 1$), according to the lexicographic sorting of the suffixes of all strings in $\mathcal{S}$. In this case we say the suffix $s_j[k, n_j + 1]$ is associated with the position i in $\mathsf{ebwt}(\mathcal{S})$. We can associate to each string $s_i \in \mathcal{S}$ a color i in $ID = \{1, 2, \ldots, m\}$. The output string $\mathsf{ebwt}(\mathcal{S})$, enhanced with the array $\mathsf{id}(\mathcal{S})$ of length N where $\mathsf{id}(\mathcal{S})[i] = r$, with $1 \leq r \leq m$ and $1 \leq i \leq N$, if $\mathsf{ebwt}(\mathcal{S})[i]$ is a symbol of the string $s_r \in \mathcal{S}$, is called *colored* EBWT.

The *longest common prefix* (LCP) array of the collection $\mathcal{S}$ [30, 18, 24] is the array $\mathsf{lcp}(\mathcal{S})$ of length $N + 1$, such that $\mathsf{lcp}(\mathcal{S})[i]$, with $2 \leq i \leq N$, is the length of the longest common prefix between the suffixes associated to the positions i and $i - 1$ in $\mathsf{ebwt}(\mathcal{S})$ and $\mathsf{lcp}(\mathcal{S})[1] = \mathsf{lcp}(\mathcal{S})[N + 1] = -1$ set by default. We denote by $\mathrm{LCP}(i, j)$ the length of the LCP between the suffixes associated with positions i and j in $\mathsf{ebwt}(\mathcal{S})$, i.e. $\min\{\mathsf{lcp}(\mathcal{S})[l] : i < l \leq j\}$. An interval $[i, j]$ with $1 \leq i < j \leq N$, is an *k-lcp interval* if $\mathsf{lcp}(\mathcal{S})[i] < k$, $\mathrm{LCP}(i, j) = k$, $\mathsf{lcp}(\mathcal{S})[j + 1] < k$. The set $\mathcal{S}$ will be later omitted if it is clear from the context.

3 Colored Longest Common Prefix Array

In this section we present a novel data structure, the *colored longest common prefix array* (cLCP). Loosely speaking, the cLCP array represents the longest common prefix between a suffix that belongs to a string of the collection $\mathcal{S}$ and the nearest suffix belonging to another string of $\mathcal{S}$, in the list of sorted suffixes of $\mathcal{S}$. In this paper, for simplicity of description, we assume that $\mathcal{S}$ is partitioned in two subsets and consider the suffixes of strings belonging to different subsets, but we remark that one can consider any situation and note also that the definition can be easily adapted for more than two sets.

For $i = 1, \ldots, N$ and $t = 1, \ldots m$, let $prev(i,t) = \max\{x \,|\, 1 \leq x < i, \mathsf{id}(\mathcal{S})[x] = t\}$ and $next(i,t) = \min\{x \,|\, i < x \leq N, \mathsf{id}(\mathcal{S})[x] = t\}$ (if such an x exists, and `null` otherwise).

In order to give the notion of the cLCP array, we first define the *Upper colored* LCP array (UcLCP) and the *Lower colored* LCP array (LcLCP), as follows.

Definition 1. *The* upper *(resp.* lower*)* colored longest common prefix *array (*UcLCP*) (resp.* LcLCP*) is a ($N \times m$)-integer array where, for each $i_r \in \{1, 2, \ldots, N\}$ with $id[i_r] = r$ and $t \in ID$,* $\mathsf{UcLCP}[i_r][t] = \mathrm{LCP}(prev(i_r,t), i_r)$ *(resp.* $\mathsf{LcLCP}[i_r][t] = \mathrm{LCP}(i_r, next(i_r,t))$*). Both* $\mathrm{LCP}(\mathtt{null}, i_r)$ *and* $\mathrm{LCP}(i_r, \mathtt{null})$ *are set equal to* 0.

Definition 2. *The* colored longest common prefix *array (*cLCP*) is a ($N \times m$)-integer array where, for each $i_r \in \{1, 2, \ldots, N\}$ with $id[i_r] = r$ and $t \in ID$,* $\mathsf{cLCP}[i_r][t] = \max(\mathsf{UcLCP}[i_r][t], \mathsf{LcLCP}[i_r][t])$.

For simplicity UcLCP, LcLCP, and cLCP are also defined when $r = t$. For all i_r such that $id[i_r] = r$, $\mathsf{UcLCP}[i_r][t]$ coincides with the correspondent value in the usual $\mathsf{lcp}(\{s_r\})$. As mentioned before, note that the notion of UcLCP, LcLCP, and cLCP can be also given for a pair of disjoint collections of strings $\mathcal{S}^0$ and $\mathcal{S}^1$ by obtaining an array defined for the pairs (i_r, t) with $id[i_r] = r$ and $t \in ID$ such that s_r and s_t belong to a different collection.

A given string $s_\chi \in \mathcal{S}^0$ having color χ implicitly induces a partition of $\mathsf{lcp}(\mathcal{S})$ in open intervals delimited by consecutive suffixes having color χ (or the position 1 and $N+1$ of lcp), called χ-*intervals*. Let us consider a position i_r contained within a χ-interval such that $\mathsf{id}[i_r] = r$ and $s_r \in \mathcal{S}^1$. Then, we can use a similar procedure as the one employed in [23], such that

$$\mathsf{UcLCP}[i_r][\chi] = \mathrm{LCP}(prev(i_r,\chi), i_r) = \min\{\mathsf{lcp}[x] : prev(i_r,\chi) < x \leq i_r\}, \quad (1)$$

$$\mathsf{LcLCP}[i_r][\chi] = \mathrm{LCP}(i_r, next(i_r,\chi)) = \min\{\mathsf{lcp}[x] : i_r < x \leq next(i_r,\chi)\}. \quad (2)$$

Additionally, we notice that there exists a relationship between the values of UcLCP related to the suffixes of s_r and the values of LcLCP related to the suffixes of s_χ. Indeed, if j_χ is a position where $\mathsf{id}[j_\chi] = \chi$, then

$$\mathsf{LcLCP}[j_\chi][r] = \mathrm{LCP}(j_\chi, next(j_\chi, r)) = \mathsf{UcLCP}[next(j_\chi, r)][\chi]. \quad (3)$$

									UcLCP			LcLCP			cLCP			
#	id	$\mathcal{S}$	ebwt	lcp	D	lcp$_\chi$	α	ζ	χ	1	2	χ	1	2	χ	1	2	Sorted suffixes of $\mathcal{S}$
1	χ	**1**	**C**	**-1**	**0**	**-1**	∞	**0**		**0**	**0**		**0**	**0**		**0**	**0**	$\$_\chi$
2	1	0	T	0	0		0	0	0			0			0			$\$_1$
3	2	0	A	0	0		0	0	0			0			0			$\$_2$
4	2	0	C	0	2		0	1	0			1			1			$A\$_2$
5	2	0	$\$_2$	1	0		0	1	0			1			1			$AACGCCGCCGGCA\$_2$
6	1	0	$\$_1$	1	4		0	3	0			3			3			$ACGAGACGAT\$_1$
7	1	0	G	4	0		0	3	0			3			3			$ACGAT\$_1$
8	2	0	A	3	5		0	4	0			4			4			$ACGCCGCCGGCA\$_2$
9	χ	**1**	$\$_0$	**4**	**0**	**0**	∞	**0**		**3**	**4**		**1**	**0**		**3**	**4**	**ACGCGCC** $\$_\chi$
10	1	0	G	1	0		1	0	1			0			1			$AGACGAT\$_1$
11	1	0	G	1	0		1	0	1			0			1			$AT\$_1$
12	χ	**1**	**C**	**0**	**2**	**0**	∞	**0**		**0**	**0**		**1**	**1**		**1**	**1**	**C** $\$_\chi$
13	2	0	G	1	0		1	0	1			1			1			$CA\$_2$
14	χ	**1**	**G**	**1**	**3**	**1**	∞	**0**		**0**	**1**		**1**	**2**		**1**	**2**	**CC** $\$_\chi$
15	2	0	G	2	0		2	0	2			1			2			$CCGCCGGCA\$_2$
16	2	0	G	3	0		2	0	2			1			2			$CCGGCA\$_2$
17	1	0	A	1	3		1	2	1			2			2			$CGAGACGAT\$_1$
18	1	0	A	3	0		1	2	1			2			2			$CGAT\$_1$
19	χ	**1**	**G**	**2**	**5**	**1**	∞	**0**		**2**	**1**		**0**	**4**		**2**	**4**	**CGCC** $\$_\chi$
20	2	0	A	4	0		4	0	4			3			4			$CGCCGCCGGCA\$_2$
21	2	0	C	5	0		4	0	4			3			4			$CGCCGGCA\$_2$
22	χ	**1**	**A**	**3**	**0**	**3**	∞	**0**		**2**	**3**		**0**	**2**		**2**	**3**	**CGCGCC** $\$_\chi$
23	2	0	C	2	0		2	0	2			0			2			$CGGCA\$_2$
24	1	0	A	0	2		0	1	0			1			1			$GACGAT\$_1$
25	1	0	C	2	0		0	1	0			1			1			$GAGACGAT\$_1$
26	1	0	C	2	0		0	1	0			1			1			$GAT\$_1$
27	2	0	G	1	3		0	2	0			2			2			$GCA\$_2$
28	χ	**1**	**C**	**2**	**4**	**0**	∞	**0**		**1**	**2**		**0**	**3**		**1**	**3**	**GCC** $\$_\chi$
29	2	0	C	3	0		3	2	3			2			3			$GCCGCCGGCA\$_2$
30	2	0	C	4	0		3	3	3			2			3			$GCCGGCA\$_2$
31	χ	**1**	**C**	**2**	**0**	**2**	∞	**0**		**1**	**2**		**0**	**1**		**1**	**2**	**GCGCC** $\$_\chi$
32	2	0	C	1	0		1	0	1			0			1			$GGCA\$_2$
33	1	0	A	0	0		0	0	0			0			0			$T\$_1$
				-1		-1												

Table 1. Let $\Sigma=\{A,C,G,T\}$, $s_\chi=ACGCGCC\$_\chi \in \mathcal{S}^0$, $s_1=ACGAGACGAT\$_1 \in \mathcal{S}^1$, and $s_2=AACGCCGCCGGCA\$_2 \in \mathcal{S}^1$. Then, $\mathsf{Score}(s_\chi, s_1)=11/7$, $\mathsf{Score}(s_\chi, s_2)=19/7$, $\mathsf{Score}(s_1, s_\chi)=15/10$, $\mathsf{Score}(s_2, s_\chi)=30/13$, and thus $\mathrm{ACS}(s_\chi, s_1)=0.67$ and $\mathrm{ACS}(s_\chi, s_2)=0.34$. In bold are all positions associated with suffixes of s_χ (i.e. the limits of the χ-intervals).

Similarly, there exists a relationship between the values of UcLCP related to suffixes of s_χ and the values of LcLCP related to suffixes of s_r. In particular,

$$\mathsf{UcLCP}[j_\chi][r] = \mathrm{LCP}(prev(j_\chi, r), j_\chi) = \mathsf{LcLCP}[prev(j_\chi, r)][\chi]. \tag{4}$$

Table 1 shows the values of UcLCP, LcLCP and cLCP of the running example, in which the collection $\mathcal{S}$ is partitioned into two subsets $\mathcal{S}^0 = \{ACGCGCC\$_\chi\}$ and $\mathcal{S}^1 = \{ACGAGACGAT\$_1, AACGCCGCCGGCA\$_2\}$.

4 Lightweight computation of the cLCP

In this section we describe a lightweight strategy to compute the colored longest common prefix array $cLCP$. For sake of simplicity we consider the case in which the collection $\mathcal{S}$ is partitioned into two subsets $\mathcal{S}^0$ and $\mathcal{S}^1$, and $\mathcal{S}^0$ consists of a single string s_χ of length n_χ. The general case can be treated analogously.

Definition 3. *A* colored k-lcp interval *is a k-lcp interval $[i, j]$ such that, among all the suffixes associated to the range $[i, j]$, at least one suffix belongs to $\mathcal{S}^0$ and at least one suffix belongs to $\mathcal{S}^1$.*

Definition 4. *Let $\mathsf{D}[1, N+1]$ denote an integer array, such that $\mathsf{D}[i] = k$ if a colored $(k-1)$-lcp interval starts at position i and for every colored h-lcp interval starting at position i then $h \leq k-1$.*

Table 1 highlights the conceptual blocks of suffixes that are associated to the positions i of D such that $\mathsf{D}[i] \neq 0$.

Note that the array D can be easily computed in $\Theta(N)$ time by linearly scanning the arrays $\mathsf{lcp}(\mathcal{S})$ and $\mathsf{id}(\mathcal{S})$ and by using a stack that simulates the computation of the colored k-lcp intervals. During the sequential scan, each element can be inserted or deleted from the stack once, at most. Furthermore, considering that each suffix could take part in no more than $\max \mathsf{lcp}(\mathcal{S})$ nested blocks, the stack requires $O(\max \mathsf{lcp}(\mathcal{S}))$ space, at most. We note that this upper bound in space is unlikely to be reached in practice, especially since the stack is emptied when two consecutive values of id corresponding to strings of different subsets are found. It is important to specify that the above mentioned stack could be stored in external memory.

In the following we describe a sequential strategy to construct cLCP of the collection $\mathcal{S}$ by using $\mathsf{id}(\mathcal{S})$, $\mathsf{lcp}(\mathcal{S})$, and $\mathsf{D}(\mathcal{S})$.

Without loss of generality, let us consider a generic string $s_r \in \mathcal{S}^1$ and $s_\chi \in \mathcal{S}^0$. Assume that $\mathsf{ebwt}[i_r]$, with $1 \leq i_r \leq N$, is associated with a suffix of s_r, i.e. $\mathsf{id}[i_r] = r \neq \chi$, and let $\chi_1 = \mathit{prev}(i_r, \chi)$ and $\chi_2 = \mathit{next}(i_r, \chi)$. Moreover, for simplicity, let UcLCP_r (resp. LcLCP_r) denote UcLCP of s_r versus s_χ (resp. LcLCP of s_r versus s_χ), i.e. the values $\mathsf{UcLCP}[i_r][\chi]$ (resp. $\mathsf{LcLCP}[i_r][\chi]$) for all such i_r; and LcLCP_χ (resp. UcLCP_χ) denote LcLCP (resp. UcLCP) of s_χ versus s_r, i.e. the values $\mathsf{LcLCP}[j_\chi][r]$ (resp. $\mathsf{UcLCP}[j_\chi][r]$) for all $1 \leq j_\chi \leq N$ such that $\mathsf{id}[j_\chi] = \chi$.

UcLCP_r *computation* — This is the easiest case, since Equation 1 allows us to directly compute UcLCP_r sequentially and linearly in the total size N of lcp. This enables us to scan lcp forward only once for all suffixes of all m strings in $\mathcal{S}^1$, by keeping track of the minimum value found since the beginning of each conceptual χ-interval (see column α in Table 1). If we consider the χ-interval (χ_1, χ_2), by employing a variable α we can iteratively compute the minimum value among consecutive elements of lcp and determine, for every $i_r \in (\chi_1, \chi_2)$, the LCP between the suffix associated with position i_r and the suffix associated with position χ_1: $\mathsf{UcLCP}[i_r][\chi] = \mathrm{LCP}(\chi_1, i_r) = \min\{\mathsf{lcp}[x] : x \in (\chi_1, i_r]\} = \alpha$.

Example 1 (Running example). If the χ-interval is $(14, 19)$ and $i_r = 18$, then $\mathsf{UcLCP}[18][\chi] = \mathrm{LCP}(14, 18) = \min\{\mathsf{lcp}[x] : x \in (14, 18]\} = \alpha[18] = 1$.

LcLCP_χ *computation* — Since LcLCP_χ is strictly related to UcLCP_r by Equation 3, we would like to compute it sequentially and linearly as well. Suppose that we have just computed $\mathsf{UcLCP}[i_r][\chi]$ and i_r represents the first suffix of s_r encountered since the beginning in (χ_1, χ_2). Then, by Equation 3, $\mathsf{LcLCP}[\chi_1][r] = \mathsf{UcLCP}[i_r][\chi]$. To keep track of the first instance of every $s_r \in \mathcal{S}^1$ in the interval, we could resort to a bit-array of m elements for χ_1.

Nevertheless, this is not sufficient to complete the construction of LcLCP_χ, because there might be no suffixes of a particular string $s_r \in \mathcal{S}^1$ within (χ_1, χ_2), but other suffixes of s_r might exist at positions $>\chi_2$. To tackle this issue, once we have thoroughly read lcp and filled LcLCP_χ using the above procedure, we can propagate the computed values of LcLCP_χ backward from lower to higher lexicographically ranked suffixes of χ, in order to complete LcLCP_χ. For example, to propagate the information from χ_2 to χ_1, we must compute:

$$\mathsf{LcLCP}[\chi_1][r] = \min\{\mathrm{LCP}(\chi_1, \chi_2), \mathsf{LcLCP}[\chi_2][r]\}. \tag{5}$$

Thus, iteratively, we can propagate the information backward from the lowest ranked suffixes of χ to the top of LcLCP_χ.

Example 2 (Running example). After the first scan of lcp in the example of Table 1, $\mathsf{LcLCP}[12][1]$ (i.e. suffix of s_χ in row 12 versus string $s_1 \in \mathcal{S}^1$) would be 0, whereas by propagating the information back from the suffix of s_χ in row 14, we obtain: $\mathsf{LcLCP}[12][1] = \min\{\mathrm{LCP}(12, 14), \mathsf{LcLCP}[14][1]\} = \min\{1, 2\} = 1$.

LcLCP_r *computation* — The most interesting part is computing LcLCP_r in such a way as to avoid the backward scan of id and lcp suggested by Equation 2 and, concomitantly, reduce the memory footprint required for keeping both UcLCP_r and LcLCP_r to a somehow negligible one. Thus, we show how to sequentially determine, for every $i_r \in (\chi_1, \chi_2)$, the LCP between the suffix associated with position i_r and the suffix associated with position χ_2.

Let us consider the array D introduced in Definition 4. Intuitively, D provides an interlacing forward information that could be exploited to compute $\mathsf{LcLCP}[i_r][\chi]$ sequentially, as soon as we reach position i_r. Firstly, observe that, for any $1 \le i_r \le N$ with $\mathsf{id}[i_r] = r$ and any $\chi_1 < x < \chi_2$, $prev(x, \chi) = prev(i_r, \chi) = \chi_1$ and $next(x, \chi) = next(i_r, \chi) = \chi_2$.

Remark 1. For any $x_1 < x_2$, with $\chi_1 \le x_1 < \chi_2$ and $\chi_1 < x_2 \le \chi_2$, $\mathrm{LCP}(x_1, x_2) \ge \mathrm{LCP}(\chi_1, \chi_2)$.

Lemma 1. *For any $1 \le i_r \le N$, if $\mathrm{LCP}(i_r, \chi_2) = k - 1$ then there exists an x, with $\chi_1 < x \le i_r$, such that $\mathsf{D}[x] = k \ne 0$ if and only if $\mathrm{LCP}(\chi_1, \chi_2) < k - 1$.*

Moreover, it follows that $\mathsf{D}[x]$ would be (the only) maximum in the range $(\chi_1, i_r]$ and its value is ≥ 2. Hence, we can determine $\mathsf{LcLCP}[i_r][\chi] = \mathrm{LCP}(i_r, \chi_2)$.

Theorem 1. *For any $1 \le i_r \le N$ such that $id[i_r] = r$, if $\mathrm{LCP}(\chi_1, i_r) > \mathrm{LCP}(\chi_1, \chi_2)$ then $\mathrm{LCP}(i_r, \chi_2) = \mathrm{LCP}(\chi_1, \chi_2)$, otherwise $\mathrm{LCP}(i_r, \chi_2) = \max\{\max\{\mathsf{D}[x] : \chi_1 < x \le i_r\} - 1, \mathrm{LCP}(\chi_1, \chi_2)\}$.*

By using Theorem 1 we need to keep track of the maximum value (decreased by 1) among consecutive D values since the beginning of each conceptual χ-interval (see column ζ in Table 1). An immediate example of Theorem 1 is given in column $\mathsf{LcLCP}[\cdot][\chi]$ of Table 1, which provides the final values of LcLCP_r, where $\mathrm{LCP}(\chi_1, i_r)$ is computed using Equation 1 and $\mathrm{LCP}(\chi_1, \chi_2)$ through $\mathsf{lcp}(\mathcal{S}^0)$ (or, shortly, lcp_χ).

Example 3 (Running example). Let $i_r = 16$ (with $prev(i_r, \chi) = 14$ and $next(i_r, \chi) = 19$) such that $\mathrm{LCP}(14, 16) = \mathsf{UcLCP}[16][\chi] = 2 > \mathrm{LCP}(14, 19) = \mathsf{lcp}_\chi[5] = 1$; then, $\mathsf{LcLCP}[17][\chi] = \mathrm{LCP}(16, 19) = \mathrm{LCP}(14, 19) = 1$. Conversely, by considering $i_r = 17$ (with $prev(i_r, \chi) = 14$ and $next(i_r, \chi) = 19$, as before), $\mathrm{LCP}(14, 17) = \mathsf{UcLCP}[17][\chi] = 1 = \mathrm{LCP}(14, 19) = \mathsf{lcp}_\chi[5] = 1$; therefore, $\mathsf{LcLCP}[17][\chi] = \mathrm{LCP}(17, 19) = \max\{\max\{\mathsf{D}[x] : 14 < x \le 17\} - 1, \mathrm{LCP}(14, 19)\} = \max\{2, 1\} = 2$. Furthermore, we consider the third case of Theorem 1 such that, for $i_r = 13$ (where $prev(i_r, \chi) = 12$ and $next(i_r, \chi) = 14$), $\mathrm{LCP}(12, 13) = \mathsf{UcLCP}[13][\chi] = 1 = \mathrm{LCP}(12, 14) = \mathsf{lcp}_\chi[4] = 1$ and thus $\mathsf{LcLCP}[13][\chi] = \mathrm{LCP}(13, 14) = \max\{\max\{\mathsf{D}[x] : 12 < x \le 13\} - 1, \mathrm{LCP}(12, 14)\} = \max\{-1, 1\} = 1$.

UcLCP_χ *computation* — Similarly to LcLCP_χ, we can compute UcLCP_χ by exploiting Equation 4 and the previously computed LcLCP_r within each χ-interval (compare columns $\mathsf{UcLCP}[\cdot][1]$ and $\mathsf{UcLCP}[\cdot][2]$ against column $\mathsf{LcLCP}[\cdot][\chi]$ in Table 1). To complete the construction of UcLCP_χ, we need then to propagate forward the information from higher to lower lexicographically ranked suffixes of χ. For example, to propagate the information from χ_1 to χ_2, we must compute $\mathsf{UcLCP}[\chi_2][r] = \min\{\mathrm{LCP}(\chi_1, \chi_2), \mathsf{UcLCP}[\chi_1][r]\}$.

To reduce the memory footprint, we can use a single matrix $\mathsf{cLCP}_\chi[1, n_\chi][1, m]$ (initialized with all 0s) to keep track of the maximum values between the corresponding positions of UcLCP_χ and LcLCP_χ, which could be then refined at most twice by propagation. Observe that UcLCP_χ, alone, can be directly computed sequentially, eventually reducing the additional space to a negligible one of size $O(m)$, as seen before for UcLCP_r and LcLCP_r.

Example 4 (Running example). After the first scan of lcp, $\mathsf{UcLCP}[22][1]$ (i.e. suffix of s_χ in row 22 versus string $s_1 \in \mathcal{S}^1$) would be 0, whereas by propagating the information forward from the suffix of s_χ at row 19, we obtain: $\mathsf{UcLCP}[22][1] = \min\{\mathrm{LCP}(19, 22), \mathsf{UcLCP}[19][1]\} = \min\{3, 2\} = 2$.

Computational complexity — The first phase of the algorithm consists of the computation of the D in $\Theta(N)$ time and $O(\max \mathsf{lcp}(\mathcal{S}))$. Notice that UcLCP_r and LcLCP_r can be determined sequentially (forward) requiring nothing but to update variables α and ζ keeping track respectively of the minimum among consecutive lcp values and of the maximum among consecutive D values since the last s_χ suffix encountered. Moreover one can observe that also in UcLCP_χ and LcLCP_χ computation both lcp_χ and cLCP_χ are actually accessed either sequentially forward or sequentially backward, up to one position before or after the currently processed one, allowing them to reside in external memory too. This means that we need $O(m)$ additional space in RAM. In order to optimally use the available size M of RAM, assuming $Q \geq 2$ is the number of m-elements rows of cLCP_χ that we could accommodate in RAM, at any moment we could just keep in memory and process only a single block of lcp_χ and cLCP_χ of size proportional to Q. Such a block, together with the bit-array of size m required in first part of LcLCP_χ computation, yield $O(mQ + \max \mathsf{lcp}(\mathcal{S}))$ overall required space (with Q a configurable parameter). Further, since cLCP_χ values could be refined at most twice by propagation, a global cost of $O(N + mn_\chi)$ time is deduced. Note that, instead, a straightforward approach that just uses Equations 1 and 2 would have required to process in RAM at least three data structures, each of size $\sim N$, using $O(n_\chi N)$ time (without propagation). In order to evaluate the I/O operations, we denote by B the disk block size and we assume that both the RAM size and B are measured in units of $(\log N)$-bit words. The overall complexity of the algorithm, included the I/O operations, is synthesized in the following theorem. The number of I/O disk operations is due the fact that the files storing the values of $\mathsf{id}(\mathcal{S})$, $\mathsf{lcp}(\mathcal{S})$, $\mathsf{D}(\mathcal{S})$, lcp_χ and cLCP_χ are used.

Theorem 2. *Let $\mathcal{S}$ a collection of m strings. Given $\mathsf{id}(\mathcal{S})[1, N]$, $\mathsf{lcp}(\mathcal{S})[1, N+1]$ and $\mathsf{lcp}(s_\chi)[1, n_\chi{+}1]$, $\mathsf{cLCP}(\mathcal{S})$ can be computed by sequential scans in $\mathcal{O}(N{+}mn_\chi)$ time and $\mathcal{O}(m{+}L_1)$ additional space, where $L_1 = \max \mathsf{lcp}(\mathcal{S})$. The total number of I/O disk operations is $O\left(\frac{1}{B \log N}(N \log m + N \log L_1 + n_\chi \log L_2 + n_\chi m \log L_1)\right)$, where $L_2 = \max \mathsf{lcp}(s_\chi)$.*

5 Multi-string ACS Computation by *cLCP*

The cLCP is a data structure that implicitly stores information useful to compute distinguishing and repeating strings in different collections. Its lightweight computation described in previous section enables the use of cLCP in several contexts in which large collections of long strings are considered.

Here, we describe its use in the case of computing the matching statistics MS and the average common substring ACS measure. The Average common substring (ACS) induced distance is based on the concept of *matching statistics* (MS) [12, 22] and is typically computed by proceeding in two steps. Let us first consider two strings s_r, of length n_r, and s_t, of length n_t, over Σ of size σ. In the first step, ACS asymmetrically computes the longest match lengths of s_r versus s_t, $\mathsf{MS}(s_r, s_t)$, where s_r is the base string. $\mathsf{MS}(s_r, s_t)[1, n_r]$ is an integer array such that, for any position j_r of s_r, $\mathsf{MS}(s_r, s_t)[j_r]$ is the length of the longest prefix of the suffix of s_r starting at position j_r that is also a substring of s_t (see Table 2). In the second step, ACS takes the average of these scores $\mathsf{Score}(s_r, s_t) = \frac{\sum_{j_r=1}^{n_r} \mathsf{MS}(s_r,s_t)[j_r]}{n_r}$; normalizes it by the lengths of s_r, s_t, and σ $Norm(\mathsf{Score}(s_r, s_t)) = \frac{\log_\sigma n_t}{\mathsf{Score}(s_r,s_t)} - \frac{2 \log_\sigma n_r}{(n_r+1)}$; and finally makes the measure symmetrical by defining $\mathrm{ACS}(s_r, s_t) = \frac{Norm(\mathsf{Score}(s_r,s_t))+Norm(\mathsf{Score}(s_t,s_r))}{2}$, in order to achieve an induced distance. We observe that ACS is not a metric, because the triangular inequality might not

$s_0[j_0]$	A	C	G	C	G	C	C			
$\mathsf{MS}(s_0, s_1)[j_0]$	3	2	1	2	1	1	1			
$s_1[j_1]$	A	C	G	A	G	A	C	G	A	T
$\mathsf{MS}(s_1, s_0)[j_1]$	3	2	1	1	1	3	2	1	1	0

Table 2. Matching statistics $\mathsf{MS}(s_0, s_1)$ and $\mathsf{MS}(s_1, s_0)$ for $s_0{=}ACGCGCC\$_0$ and $s_1{=}ACGAGACGAT\$_1$ on $\Sigma = \{A, C, G, T\}$. It follows that $\mathsf{Score}(s_0, s_1) = 11/7$, $\mathsf{Score}(s_1, s_0) = 15/10$ and, thus, $\mathrm{ACS}(s_0, s_1) = 0.67$.

hold in general. Nevertheless, if we assume s_r and s_t be generated by finite-state Markovian probability distributions, it follows that ACS is a natural distance measure between these distributions [34].

For simplicity we assume that we have a set consisting of only one string $\mathcal{S}^0 = \{s_\chi\}$, of length n_χ, and a set of strings $\mathcal{S}^1 = \{s_1, s_2, \ldots, s_m\}$, of length $N^1 = \sum_{1 \le r \le m} n_r$, and we want to compute the pairwise ACS induced distances between $\mathcal{S}^0$ (or, more explicitly, s_χ) and every other string in $\mathcal{S}^1$ simultaneously, as in the multi-string ACS problem. Our approach could be also applied to a more general case.

There is a clear correspondence between cLCP of s_χ versus all strings in $\mathcal{S}^1$, and MS of ACS. More precisely:

Proposition 1. *Given any two strings $s_r, s_\chi \in \mathcal{S}$, $\mathsf{MS}(s_r, s_\chi)$ is a permutation of all values in $\mathsf{cLCP}(\mathcal{S})$ related to the suffixes of s_r (the base string) versus s_χ: $\mathsf{MS}(s_r, s_\chi)[j_r] = \mathsf{cLCP}[i_r][\chi]$, where $1 \le i_r \le N$ such that $\mathsf{id}(\mathcal{S})[i_r] = r$, and j_r is the starting position in s_r of the suffix associated with $\mathsf{ebwt}(\mathcal{S})[i_r]$.*

Indeed, for each suffix of every string $s_r \in \mathcal{S}^1$, associated with $\mathsf{ebwt}[i_r]$, $\mathsf{cLCP}[i_r]$ would account for the longest prefix that is a substring of s_χ, and this must correspond to one of the closest suffixes belonging to s_χ immediately above ($prev(i_r, \chi)$) or below ($next(i_r, \chi)$) row i_r in the sorted suffixes list.

We can thus exploit the above proposition to compute MS using cLCP, by using the strategy described in previous section. In fact, computing MS by straightly using the Equations 1 and 2 would require to explicitly keep track of cLCP for each χ-interval, which could have width $\Theta(N)$ in the worst case. In this section we show that this additional space can be controlled and reduced by using our lightweight computation of cLCP.

Using the construction described in Section 4 we can determine UcLCP_r and LcLCP_r sequentially (forward) and these values are definitive (they are not subject to refinement by propagation). Therefore we can reduce multi-string ACS memory footprint by summing up all the maximum values between the respective positions in UcLCP_r and LcLCP_r for every specific string $s_r \in \mathcal{S}^1$, and for every position i_r, and storing them into an array Score_r of size m as they are computed during forward phase, without explicitly maintaining cLCP_r values in internal or external memory. On the other hand, since UcLCP_χ and LcLCP_χ require propagation to be completed, we need to maintain (a Q-sized portion of) cLCP_χ matrix and similarly cumulate cLCP_χ values for every position j_χ and for every string $s_r \in \mathcal{S}^1$ into an array Score_χ of size m as these values became definitive during backward phase. Accordingly, multi-string ACS computation does not add to cLCP construction more than $\Theta(m)$ space and $O(mn_\chi)$ time. Note that in a typical application, m can be assumed $\ll N$ and negligible compared to the internal memory available. Here, we show a simplified version of our strategy described in Section 5. For simplicity, we index the files as array, but the reader can note that we only access to them sequentially. We need to keep in memory the length of the strings for the m ACS scores.

6 Preliminary Experiments

As a proof-of-concept, we test our new data structure (cLCP) with a prototype tool named cLCP-mACS [1] designed to solve the multi-string ACS problem. All tests were done on a DELL PowerEdge R630 machine, 24-core machine with Intel(R) Xeon(R) CPU E5-2620 v3 at 2.40 GHz, with 128 GB of shared memory, used in not exclusive mode. The system is Ubuntu 14.04.2 LTS.

We show that our preliminary experiments confirm the effectiveness of our approach for the multi-string ACS problem, that consists in the pairwise comparison of a single string against a set of m strings simultaneously, in order to obtain m ACS induced distances. This is not a limitation, because the computation of pairwise distances between strings of a collection $\mathcal{S}$ can be treated analogously, in the sense that one could execute our tool more times, without computing the data structures of the preprocessing step.

We experimentally observed that the preprocessing step is more computationally expensive than the step for computing the m ACS distances via cLCP. The problem of computing the $\mathsf{ebwt}(\mathcal{S}), \mathsf{lcp}(\mathcal{S}), \mathsf{id}(\mathcal{S})$ are being intensively studied, and improving its efficiency is out of the aim of this paper. So, we omit time/space requirements of the preprocessing step, since (i) these data structures can be reused and (ii) different programs [2–5] are used to construct them. So, we solely focus on the phase of computation of the matrix distances.

To assess the performance of our algorithm we consider the two collections of genomes listed in [1] and described in Table 3.

The auxiliary disk space for the first collection is 34 Gbytes and 108 Gbytes for the second one. But, note that the D file is sparse, in the sense that it contains many zero-values. One can reduce its size by storing only non-zero values or using a different encoding of these values, for instance Sadakane's [32] encoding. However, this could slow down the computation.

Notice that an entirely like-for-like comparison between our implementation and the above existing implementation is not possible, since, for the best of our knowledge, our tool is the first lightweight tool.

The ACS computation of two strings has been faced in [34] and it has been implemented by using the k Mismatch Average Common Substring Approach (kmacs) tool [6]. For $k = 0$ kmacs exactly computes the ACS. Other algorithms besides kmacs [33, 29] have been designed to compute alignment-free distances based on longest matches with mismatches, but for the special case $k = 0$ kmacs

	$\vert\mathsf{ebwt}(\mathcal{S})\vert$	Min length	Max length	Max lcp	Program	Wall clock	Memory
	(Gbytes)					(mm:ss)	(kbytes)
1	3.434	1,080,084	10,657,107	1,711,194	cLCP-mACS	27:42	6,160
					kmacs	18:50	3,723,728
2	9.258	744	14,782,125	5,714,157	cLCP-mACS	53:03	6,372
					kmacs	58:08	14,717,332

Table 3. The first collection contains 932 genomes, the second one contains $4,983$ genomes. Note that $|s_\chi| = 5,650,368$ for the first collection and $|s_\chi| = 3,571,103$ for the second one. In both cases these values are greater than the average length of the strings in the respective collection. The amount of time elapsed from the start to the completion of the instance. The column memory is the peak Resident Set Size (RSS). Both values were taken with `/usr/bin/time` command.

is the software that has the better change to scale with the dataset size. The implementation of kmacs works in internal memory and not in sequential way. More in detail, it works in m steps, at each step it builds the suffix array [25] and the lcp array of two strings s_i and s_j (for $1 \leq i < j \leq m$) in order to compute the ACS distance between s_i and s_j. In our experiments we have fixed $s_\chi = s_1$, so we compare s_χ with s_j (for $2 \leq j \leq m$). Note that the performance in terms of time of kmacs could be improved by separately considering the computation of the auxiliary data structures. However, the occupation of RAM would remain almost the same.

The experimental results in Table 3 show that our algorithm is a competitive tool for the lightweight simultaneous computation of a pairwise distance of a string versus each string in a collection, in order to obtain an efficient solution for the multi-string ACS problem.

7 Conclusions and Further works

We have introduced the colored longest common prefix array (cLCP) that, given a strings collection $\mathcal{S}$, stores the length of the longest common prefix between the suffix of any string in $\mathcal{S}$ and the nearest suffix of another string of $\mathcal{S}$, according to the lexicographically sorted list of suffixes. This notion is extended in a natural way considering two collections of strings $\mathcal{S}^0$ and $\mathcal{S}^1$ in order to consider the longest common prefix between any pair of strings in different collections. We have also provided a lightweight method that computes cLCP via sequential scans when $\mathcal{S}^0$ consists of a single string, but the strategy can be applied to the general case. This fact makes this data structure appropriate for computing several types of statistics on large collections of long strings. In particular, we have proved that $\mathsf{cLCP}(\mathcal{S})$ produces a permutation of the matching statistics (MS) for the strings of the collection of $\mathcal{S}$. Based on this result, we have used cLCP to efficiently solve the multi-string ACS problem — i.e. computing pairwise MS between a string in $\mathcal{S}^0$ and all m strings in $\mathcal{S}^1$ simultaneously, — that is nowadays crucial in many practical applications, but demanding for large string comparisons. This is also supported by experimental results.

It is interesting to note that the data structure proposed in this paper, and its sequential strategy of computation, are intrinsically dynamic, i.e. cLCP can be efficiently updated when the collection is modified by inserting or removing a string. In particular, after the removal of a string, cLCP can be updated by exploiting the mathematical properties of the permutation associated with the EBWT. The insertion of a new string in the collection can be managed by using merging strategy proposed in [20] that works in semi-external memory. In this case the array D can be constructed during this merging phase. Finally, we plan to extend our framework to solve the many-to-many pairwise ACS problem on a collection $\mathcal{S}$ of m sequences or between all strings of a collection versus all strings of another collection at the same time.

References

1. https://github.com/giovannarosone/cLCP-mACS
2. https://github.com/BEETL/BEETL
3. https://github.com/giovannarosone/BCR_LCP_GSA
4. https://github.com/felipelouza/egsa
5. https://github.com/felipelouza/egap
6. http://kmacs.gobics.de/
7. Apostolico, A., Guerra, C., Pizzi, C.: Alignment Free Sequence Similarity with Bounded Hamming Distance. In: Data Compression Conference, DCC 2014. pp. 183–192. IEEE (2014)
8. Bauer, M., Cox, A., Rosone, G.: Lightweight algorithms for constructing and inverting the BWT of string collections. Theor. Comput. Sci. **483**(0), 134–148 (2013)
9. Belazzougui, D., Cunial, F.: Indexed matching statistics and shortest unique substrings. In: String Processing and Information Retrieval. pp. 179–190. Springer International Publishing (2014)
10. Belazzougui, D., Cunial, F.: Fast label extraction in the cdawg. In: String Processing and Information Retrieval. pp. 161–175. Springer International Publishing (2017)
11. Burrows, M., Wheeler, D.: A block sorting data compression algorithm. Tech. rep., DEC Systems Research Center (1994)
12. Chang, W.I., Lawler, E.L.: Sublinear approximate string matching and biological applications. Algorithmica **12**(4), 327–344 (1994)
13. Cohen, E., Chor, B.: Detecting phylogenetic signals in eukaryotic whole genome sequences. J. Comput. Biol. **19**(8), 945–956 (2012)
14. Comin, M., Verzotto, D.: The Irredundant Class method for remote homology detection of protein sequences. J. Comput. Biol. **18**(12), 1819–1829 (2011)
15. Comin, M., Verzotto, D.: Alignment-free phylogeny of whole genomes using underlying subwords. Algorithms Mol. Biol. **7**(1) (2012)
16. Comin, M., Verzotto, D.: Whole-genome phylogeny by virtue of unic subwords. In: DEXA. pp. 190–194. IEEE (2012)
17. Comin, M., Verzotto, D.: Beyond fixed-resolution alignment-free measures for mammalian enhancers sequence comparison. IEEE/ACM Trans. Comput. Biol. Bioinform. **11**(4), 628–637 (2014)
18. Cox, A.J., Garofalo, F., Rosone, G., Sciortino, M.: Lightweight LCP construction for very large collections of strings. J. Discrete Algorithms **37**, 17–33 (2016)

19. Cox, A.J., Jakobi, T., Rosone, G., Schulz-Trieglaff, O.B.: Comparing DNA sequence collections by direct comparison of compressed text indexes. In: WABI. LNCS, vol. 7534 LNBI, pp. 214–224. Springer (2012)
20. Egidi, L., Louza, F.A., Manzini, G., Telles, G.P.: External memory BWT and LCP computation for sequence collections with applications. ArXiv e-prints (2018)
21. Ferraro Petrillo, U., Guerra, C., Pizzi, C.: A new distributed alignment-free approach to compare whole proteomes. Theor. Comput. Sci. **698**, 100–112 (2017)
22. Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press (1997)
23. Leimeister, C.A., Morgenstern, B.: kmacs: the *k*-mismatch average common substring approach to alignment-free sequence comparison. Bioinformatics **30**(14), 2000–2008 (2014)
24. Louza, F., Telles, G., Hoffmann, S., Ciferri, C.: Generalized enhanced suffix array construction in external memory. Algorithms Mol. Biol. **12**(1), 26 (2017)
25. Manber, U., Myers, G.: Suffix Arrays: A New Method for On-line String Searches. In: Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 319–327. SODA '90, Society for Industrial and Applied Mathematics (1990)
26. Mantaci, S., Restivo, A., Rosone, G., Sciortino, M.: An extension of the Burrows-Wheeler Transform. Theor. Comput. Sci. **387**(3), 298–312 (2007)
27. Mantaci, S., Restivo, A., Rosone, G., Sciortino, M.: A new combinatorial approach to sequence comparison. Theory Comput. Syst. **42**(3), 411–429 (2008)
28. Ohlebusch, E., Gog, S., Kügell, A.: Computing matching statistics and maximal exact matches on compressed full-text indexes. In: SPIRE. LNCS, vol. 6393, pp. 347–358. Springer-Verlag (2010)
29. Pizzi, C.: MissMax: alignment-free sequence comparison with mismatches through filtering and heuristics. Algorithms for Molecular Biology **11**, 6 (2016)
30. Puglisi, S., Turpin, A.: Space-time tradeoffs for longest-common-prefix array computation. In: ISAAC, LNCS, vol. 5369, pp. 124–135. Springer (2008)
31. Ren, J., Song, K., Sun, F., Deng, M., Reinert, G.: Multiple alignment-free sequence comparison. Bioinformatics **29**(21), 2690–2698 (2013)
32. Sadakane, K.: Compressed suffix trees with full functionality. Theor. Comp. Sys. **41**(4), 589–607 (2007)
33. Thankachan, S., Chockalingam, S., Liu, Y., Apostolico, A., Aluru, S.: ALFRED: a practical method for alignment-free distance computation. J. Comput. Biol. **23**(6), 452–460 (2016)
34. Ulitsky, I., Burstein, D., Tuller, T., Chor, B.: The average common substring approach to phylogenomic reconstruction. J Comput. Biology **13**(2), 336–350 (2006)
35. Zielezinski, A., Vinga, S., Almeida, J., Karlowski, W.: Alignment-free sequence comparison: benefits, applications, and tools. Genome Biol. **18**(1), 186 (2017)