Sorting suffixes via inverse Lyndon factorization

Paola Bonizzoni$^{1[0000-0001-7289-4988]}$, Raffaella Rizzi$^{1[0000-0001-9730-7516]}$, Rocco Zaccagnino$^{2[0000-0002-9089-5957]}$, and Rosalba Zizza$^{2[0000-0001-9144-3074]}$

^{1} Dip. di Informatica, Sistemistica e Comunicazione, University of Milano-Bicocca, viale Sarca 336, 20126 Milan, Italy
{paola.bonizzoni, raffaella.rizzi}@unimib.it

^{2} Dip. di Informatica, University of Salerno, via Giovanni Paolo II 132, 84084 Fisciano, Italy
{rzaccagnino, rzizza}@unisa.it

Abstract. In this paper we exploit how some theoretical properties of the Inverse Lyndon Factorization of a string S, denoted $\mathsf{ICFL}(S)$, can be used for sorting the suffixes of S, *i.e.*, for computing its suffix array. Briefly, $\mathsf{ICFL}(S)$ was introduced in [8] as a factorization of S in a strictly increasing lexicographic sequence of inverse Lyndon words, *i.e.*, strings which are greater than any of their proper nonempty suffixes. $\mathsf{ICFL}(S)$ is unique and can be computed in linear time [8]. Here, we focused on the *compatibility property* [8], counterpart for $\mathsf{ICFL}(S)$ of the property proved for the Lyndon Factorization $\mathsf{CFL}(S)$ in [18, 19]. Shortly, this property induces the sorting of the suffixes of S ("global suffixes") from the sorting of the suffixes of the factors of $\mathsf{ICFL}(S)$ ("local suffixes"). In addition, we also use a bound proved in [10] for the longest common prefix between two substrings of S. Here, we show how we can take advantage of these two results for inducing the sorting of the suffixes of S.

1 Introduction

It is well known that the *Suffix Array* (SA) of a text S, which stores the starting positions of the suffixes of S in increasing lexicographic order, can be computed in linear time [15]. On the other hand, the wide usage of this data structure for practical applications (e.g., in Bioinformatics and Data Compression) necessitates new investigations for producing increasingly efficient algorithms for practical applications [1]. As written in [4], the best practical algorithm was proposed by Yuta Mori, never published but described in [14] (see also [15] for other references). A new approach for sorting suffixes, using the properties of *Lyndon words*, is presented in [2]. A Lyndon word is a string which is strictly lexicographically smaller than each of its proper cyclic shifts, or, equivalently, than each of its nonempty proper suffixes [17]. As mentioned in [2], even though the proposed algorithm is not competitive in practice, it opened a new perspective from a theoretical point of view. Indeed, few years later special properties of Lyndon words have been used for accelerating suffix sorting in practice [4]. In both papers [2] and [4], the general idea is to separate suffixes of S in groups, sort the suffixes inside each group and then, by using properties of Lyndon words, merge the groups to obtain the sorting of the suffixes of S. This idea was already proposed some years before in [19], starting from the Lyndon Factorization of S, a factorization introduced by Chen, Fox, Lyndon in [12] and so named $\mathsf{CFL}(S)$. The $\mathsf{CFL}(S)$ is the unique factorization of S in a lexicographically non-increasing sequence of factors which are Lyndon words. It can be computed in linear time [13]. The *Compatibility Property* proved in [19] may be used for obtaining the sorting of the suffixes of S ("global suffixes") by using the sorting of the suffixes of each factor of $\mathsf{CFL}(S)$ ("local suffixes"). However, as the authors said, the key aspect of this approach lies in finding an efficient procedure for obtaining the sorting of the global suffixes from the sorting of the local suffixes.

Here, we continue the investigation of the idea of using the sorting of suffixes of the factors of a Lyndon factorization, by proposing a novel approach for computing the Suffix Array of S exploiting the properties of a new Lyndon based factorization, called *Canonical Inverse Lyndon Factorization* of S, named $\mathsf{ICFL}(S)$. We point out that its practical potential has been demonstrated [11] in the detection of overlaps between genomic reads obtained from sequencing. Additionally, it has been utilized in [5–7, 11] in assigning transcriptomic (RNA-Seq) reads to their respective origin gene. While the above applications of $\mathsf{ICFL}(S)$ are based on a property of $\mathsf{ICFL}(S)$ called *conservation of factors*, properties related to the longest common prefix of substrings of S proved in [8–10] strongly suggest that $\mathsf{ICFL}(S)$ may be used to develop new algorithms for the Suffix Array computation. Here, our contribution is to shed light on how the above novel theoretical properties of ICFL can be used in the development of a new class of good practical algorithms for computing the Suffix Array; such a computation relies on using properties of the factorization of a string to merge sorted local suffixes of factors, typically much shorter than the string itself, with the goal of sorting its global suffixes.

Recall that $\mathsf{ICFL}(S)$ was introduced in [8], as a factorization composed of a strictly lexicographically increasing sequence of *inverse Lyndon words*, *i.e.*, strings which are greater than any of their proper nonempty suffixes (see [8] for a deep investigation about the relation between inverse Lyndon words and other similar notions derived from Lyndon words). $\mathsf{ICFL}(S)$ maintains the main properties of $\mathsf{CFL}(S)$: it is unique for S and can be computed in linear time. In addition, it maintains a similar Compatibility Property (Lemma 1). Most notably, $\mathsf{ICFL}(S)$ has another interesting property [8–10], that can be useful in order to design an effective implementation of our strategy: thanks to $\mathsf{ICFL}(S)$, we can provide an upper bound on the length of the longest common prefix of two substrings of S starting from different positions on S (Proposition 1). The proof of Proposition 1 is done thanks to another property of $\mathsf{ICFL}(S)$ which states that the longest common prefix between two consecutive factors of $\mathsf{ICFL}(S)$ is shorter than the border of the first factor, where let us recall that the border of a string S is the longest prefix of S that is also a suffix of S. It is worthy of note that this property is not true in general for a factorization in which the unique prerequisite is the strictly increasing order of its factors [10] and, clearly, cannot be proved for $\mathsf{CFL}(S)$, being each factor an unborderd word.

After a preliminary section introducing basic notions on strings and ICFL, in Section 3 we present our strategy for constructing the Suffix Array of a text S by using its ICFL factorization. Work in progress and challenging open questions are discussed in a final section (Section 4). Omitted proofs are reported in the Appendix.

2 Preliminary notions and known results

In the following, Σ is a totally ordered alphabet. A string $S \in \Sigma^*$ of length n over Σ is a finite sequence $a_0 a_1 \cdots a_{n-1}$ of n characters $a_i \in \Sigma$, $i \in \{0, \dots, n-1\}$ and ϵ denotes the empty string. For a string $S = a_0 a_1 \cdots a_{n-1}$, we denote by $|S| = n$ its length. Let $i, j \in \{0, \dots, n-1\}$. We denote by $S[i]$ the character of S at *position* i, *i.e.*, the $(i+1)$-th character of S, and by $S[i, j]$ the substring of S starting with $S[i]$ and ending with $S[j]$. If $i = 0$ (resp. $j = n-1$), $S[0, j]$ (resp. $S[i, n-1]$) is a prefix of S (resp. a suffix of S) and when $j \neq n-1$ (resp. $i \neq 0$) the prefix $S[0, j]$ (resp. suffix $S[i, n-1]$) is proper. We denote by S_i the suffix of S starting at position i, *i.e.*, $S_i = S[i, n-1]$. The set of the nonempty suffixes (resp. prefixes) of S will be denoted by $Suff(S)$ (resp. $Pref(S)$). Furthermore, we say that two strings S and S' are in *prefix relation* if either S is a prefix of S' or S' is a prefix of S. We recall that, given a nonempty string S, *a border* of S is a string which is both a proper prefix and a proper suffix of S. The longest border is also called *the border* of S. A nonempty string $S \in \Sigma^*$ is *bordered* if it has a nonempty border, otherwise, S is *unbordered*. A nonempty string S is *primitive* if $S = X^k$, $X \in \Sigma^*$, implies $k = 1$. An unbordered string is primitive [16].

By natural extension of the lexicographic order $<$ from characters to strings, given $S, S' \in \Sigma^*$, we have $S < S'$ (resp. $S \leq S'$), *i.e.*, S is *strictly smaller* than S' (resp. smaller than or equal to S'), if either S is a proper prefix (resp. prefix) of S' or $S = XaY, S' = XbZ$, $a < b$, for $a, b \in \Sigma$ and $X, Y, Z \in \Sigma^*$. Furthermore, for two nonempty strings S, S', we write $S \ll S'$ if $S < S'$ and additionally S is not a proper prefix of S' [3].

A *factorization* of a string S is a sequence $F(S) = \langle f_1, f_2, \dots, f_k \rangle$ of nonempty strings $f_1, f_2, \dots, f_k$ such that $S = f_1 f_2 \cdots f_k$ [16]. In the sequel, the term *factor* is only used for identifying each string $f_1, f_2, \dots, f_k$ in $F(S) = \langle f_1, f_2, \dots, f_k \rangle$, for a given S and F. The *Canonical Inverse Lyndon factorization* $\mathsf{ICFL}(S) = \langle f_1, f_2, \dots, f_k \rangle$ was introduced in [8] as a special factorization of S such that $f_1 \ll f_2 \ll \cdots \ll f_k$ and each f_i is an *inverse Lyndon word*, that is, each nonempty proper suffix of f_i is *strictly smaller* than f_i, $1 \leq i \leq k$. For example, *cac* and *tcaccgc* over $\Sigma = \{a, c, g, t\}$ and with $a < c < g < t$, are inverse Lyndon words. In fact, the suffix *c* is prefix of *cac* and the suffix *ac* is lexicographically smaller than the entire string *cac*; moreover, in the second example, no suffix of *tcaccgc* neither occurs as prefix of the string nor is lexicographically smaller. Observe that, differently from Lyndon words, inverse Lyndon words can be bordered. Let us consider the string $S = gcatcaccgctctacagaac$. We have that $\mathsf{ICFL}(S) = \langle gca, tcaccgc, tctacagaac\rangle$.

We now recall the two fundamental results on $\mathsf{ICFL}(S)$ which are used by our strategy. Let x, $first(x)$ and $last(x)$ be a nonempty substring of S and its starting and ending positions on S, respectively. The substring $S[i, last(x)]$, such that $first(x) \leq i \leq last(x)$, is a suffix of x and will be denoted by $\overline{S}_{i,x}$. Note that the suffix $S_i = S[i, n-1]$ has clearly $\overline{S}_{i,x}$ as prefix. In the following, S_i and $\overline{S}_{i,x}$ will be called respectively *global suffix* (*w.r.t.* S) and *local suffix* (*w.r.t.* x) starting at i.

The next lemma is the first result we use in our strategy: the *local suffixes w.r.t.* substrings detected by ICFL factorization keep their mutual order when extended to the suffixes of the whole word (*global suffixes*).

Lemma 1. *[9, 10]* (Compatibility Property) *Let $S \in \Sigma^*$ be a nonempty string which is not an Inverse Lyndon word. Let $\mathsf{ICFL}(S) = \langle f_1, \ldots, f_k\rangle$ and $x = f_i f_{i+1} \cdots f_h$, such that $1 \leq i \leq h \leq k$. Let us assume that $\overline{S}_{j_1,x} < \overline{S}_{j_2,x}$, where $first(x) \leq j_1 \leq last(x)$, $first(x) \leq j_2 \leq last(x)$, $j_1 \neq j_2$. If $\overline{S}_{j_1,x}$ is a proper prefix of $\overline{S}_{j_2,x}$ and $h < k$, then $S_{j_2} < S_{j_1}$, otherwise $S_{j_1} < S_{j_2}$.*

Example 1. As an example, let $S = aaabcaabcadcaabca$. We have $\mathsf{ICFL}(S) = \langle aaa, b, caabca, dcaabca\rangle$. Choose $x = aaabcaabca$. Consider the local suffixes $\overline{S}_{1,x} = aabcaabca$ and $\overline{S}_{5,x} = aabca$. We have that $\overline{S}_{5,x} = aabca < aabcaabca = \overline{S}_{1,x}$, being $\overline{S}_{5,x}$ is a prefix of $\overline{S}_{1,x}$. By Lemma 1, given the two global suffixes $S_1 = aabcaabcadcaabca$ and $S_5 = aabcadcaabca$, We have that $S_1 < S_5$. Now, given the local suffix $\overline{S}_{6,x} = abca$, we have that $\overline{S}_{1,x} = aabcaabca < abca = \overline{S}_{6,x}$ but $\overline{S}_{1,x} = aabcaabca$ is not a prefix of $abca = \overline{S}_{6,x}$. Thus by Lemma 1 given the two global suffixes $S_1 = aabcaabcadcaabca$ and $S_6 = abcadcaabca$, we have that $S_1 < S_6$. For a more deep insight about this property, refer to [8, 10].

In the sequel, we denote $\overline{S}_{i,x}$ with $\overline{S}_i$ when x is a factor of $\mathsf{ICFL}(S)$. Notice that in this case i is a position of S such that $first(x) \leq i \leq last(x)$, that is $\overline{S}_i$ denotes the local suffix of the factor of S that includes position i of S. Also, we will use the term *local suffix* to refer only to a suffix of a factor of $\mathsf{ICFL}(S)$.

Let X and Y be two strings, we denote by $\mathrm{lcp}(X, Y)$ and $\mathrm{LCP}(X, Y)$ the *longest common prefix* of X and Y, and its *length*, respectively. Then, given $\mathsf{ICFL}(S) = \langle f_1, f_2, \ldots, f_k\rangle$, and let $\mathcal{M} = \max\{|f_i f_{i+1}| \mid 1 \leq i < k\}$ be the maximum length of two consecutive factors of $\mathsf{ICFL}(S)$, the Proposition 1 holds (see Proposition 7.2 [10]).

Proposition 1. *[10] Let $S \in \Sigma^*$ be a nonempty string which is not an inverse Lyndon word and let $\mathsf{ICFL}(S) = \langle f_1, \ldots, f_k\rangle$. For each nonempty proper substrings X, Y of S starting from different positions on S, we have* $\mathrm{LCP}(X, Y) = |\mathrm{lcp}(X, Y)| \leq \mathcal{M}$.

The above proposition states the second fundamental result exploited by our strategy. As detailed in [10], this result was proved thanks to another property, that holds for $\mathsf{ICFL}(S)$, but not for $\mathsf{CFL}(S)$. In this paper we show how we can use this bound for sorting suffixes of a strings by sorting substrings of a bounded length, when X, Y are suffixes of S.

3 The strategy

The proposed strategy for computing the Suffix Array, based on the use of properties of ICFL, is characterized by the following schema.

ICFL-based Algorithmic Schema for Suffix Array computation

`Step1:` *compute, for each distinct local suffix s, a* **ranking list** *containing all the occurrences (starting positions) of s as a local suffix, listed by lexicographic order of the global suffixes starting with s.*

`Step2:` *construct* **chains** *of local suffixes in prefix relation and obtains a similar ranking list for each chain.*

`Step3:` *merge the ranking lists of the chains to build the Suffix Array of S.*

3.1 Step1: compute the ranking lists for all distinct local suffixes

In our framework, the *ranking list* of a local suffix s is the permutation of its occurrences in S as a local suffix, according to the lexicographic order of the global suffixes starting with s as prefix. Below the formal definition of the ranking list is stated by means of two main conditions.

Definition 1. *Let s be a local suffix of a factor of $\mathsf{ICFL}(S) = \langle f_1, \ldots, f_k \rangle$. The ranking list of s (also called $[s]$*`-ranking`*) is the sequence of its distinct (starting) positions $[e_1, \ldots, e_m]$ such that:*

(i) $S[e_i, e_i + |s| - 1] = s$ *for each* $i = 1, \ldots, m-1$*, and there exists a factor* f_h *of* $\mathsf{ICFL}(S)$ *such that* $f_h = x'S[e_i, e_i + |s| - 1]$*,* $x' \in \Sigma^*$*,* $h \in \{1, \ldots, k\}$*,*
(ii) $S_{e_i} < S_{e_{i+1}}$ *for each* $i = 1, \ldots, m-1$*.*

It is easy to see that the ranking list is unique, for each local suffix. Table 2 reports the ranking lists for each local suffix of the ICFL represented in Table 1. As an example, the position 4 does not belong to $[ca]$`-ranking` since ca does not occur in position 4 as a local suffix (condition *(i)*). For the same reason, position 2 does not belong to $[abca]$`-ranking`.
Step1. The ranking lists of all the (distinct) local suffixes are computed as follows:

- the factors are considered starting from f_{k-1} (the second last one) down to f_1 (the first one) and finally f_k is considered (the last one). Observe that a local suffix occurring in f_k corresponds to a (global) suffix of S;
- let f_j be the currently considered factor. For each position i in f_j, let $s = \overline{S}_{i,f_j} = \overline{S}_i$ be a suffix f_j, *i.e.*, a local suffix. Then, i is always inserted at the beginning of the initially empty list $[s]$`-ranking`.

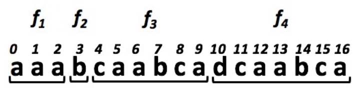

Table 1: ICFL factorization of $S = aaabcaabcadcaabca$

Local Suffix s	**[s]-ranking]**
a	$[16, 2, 9]$
aa	$[1]$
aaa	$[0]$
$aabca$	$[12, 5]$
$abca$	$[13, 6]$
b	$[3]$
bca	$[14, 7]$
ca	$[15, 8]$
$caabca$	$[11, 4]$
$dcaabca$	$[10]$

Table 2: Ranking lists

The correctness of the ranking lists computed as described above is a direct consequence of previously stated Lemma 1 and is proved by the next proposition (the proof is reported in the Appendix).

Proposition 2. *Let s be a local suffix. The sequence $L_s = [e_1, \ldots, e_m]$ constructed by Step1 is the $[s]$-*`ranking`*.*

3.2 Step2: compute the ranking lists of the prefix chains

Starting from the ranking lists computed by Step1 for the distinct local suffixes, Step2 builds the ranking lists of the chains of local suffixes in prefix relation. For this purpose, we introduce the notion of *suffix prefix chain.*

Definition 2. *A sequence $C = [s_1, \ldots, s_n]$ of distinct local suffixes $s_1, \ldots, s_n$ is a* `suffix prefix chain` *if*

(i) *s_i is prefix of s_{i+1}, for each $i = 1, \ldots, n-1$,*
(ii) *there is no another local suffix y, not belonging to C, which is a prefix of s_i or s_i is a prefix of y, for each $i = 1, \ldots, n$.*

Note that (ii) is a maximality condition of a suffix prefix chain C. For example, the sequence $[a, aabca]$ is not a suffix prefix chain, since the local suffix aa does not belong to $[a, aabca]$, but aa is a prefix of $aabca$ and has a as a prefix. We can easily check that the sequence $[a, aa, aabca]$ is suffix prefix chain.

In this context, we consider the notion of C-`ranking`, with C a suffix prefix chain, as a natural extension of $[s]$-`ranking`, with s a local suffix. It is rather straightforward, but it is reported below for the sake of clarity.

Definition 3. *Let $\mathsf{ICFL}(S) = \langle f_1, \ldots, f_k \rangle$ and let $C = [s_1, \ldots, s_n]$ be a suffix prefix chain. Then $[e_1, \ldots, e_m]$, $e_i \neq e_t$, $i, t \in \{1, \ldots, m\}$, is the ranking list of C (also called C-*`ranking`*) if*

(i) *for each* $i = 1, \ldots, m$*, there exists a local suffix* s_j *in* C *such that* $S[e_i, e_i + |s_j| - 1] = s_j$ *and there exists a factor* f_h *of* $\mathsf{ICFL}(S)$*, such that* $f_h = x' S[e_i, e_i + |s_j| - 1]$*,* $x' \in \Sigma^*$*,* $h \in \{1, \ldots, k\}$*,*

(ii) *for each* $j = 1, \ldots, n$*, all the positions in* $[s_j]$`-ranking` *are contained in* $[e_1, \ldots, e_m]$*,*

(iii) $S_{e_i} < S_{e_{i+1}}$ *for each* $i = 1, \ldots, m - 1$*.*

Shortly, the ranking list $[e_1, \ldots, e_m]$ of a suffix prefix chain $C = [s_1, \ldots, s_n]$ is the merge without repetitions of the ranking lists of the local suffixes s_j for $j = 1, \ldots, n$ (conditions (i) and (ii)) such that the sorting of the corresponding global suffixes is preserved (condition (iii)). For example, $[16, 13, 6, 9]$ is not the $[a, abca]$`-ranking`, since we forgot 2 that belongs to the $[a]$`-ranking`. Instead, $[a, abca]$`-ranking` $= [16, 13, 2, 6, 9]$. As another example, $[16, 1, 0, 2, 9]$ is not the $[a, aa, aaa]$`-ranking` since $S_0 < S_1$. Instead, $[a, aa, aaa]$`-ranking` $= [16, 0, 1, 2, 9]$.

To describe how to compute the suffix prefix chains and their ranking lists, it could be useful to "visualize" the local suffixes in a data structure that can facilitate understanding of how these can be processed and combined to obtain the final suffix array. The data structure chosen for this purpose is the *Prefix Tree*. In our tree, each node, except the root, represents a distinct local suffix. The root represents the empty word, a dummy node. The main rule governing connections between nodes is the following: given a node, the local suffix represented by such a node has as a prefix the local suffix represented by the parent node. Figure 1 shows the prefix tree for the example depicted in Tables 1-2. Observe that, in such a tree, the sequence of local suffixes represented by the nodes that we find in the path that goes from a child of the root to any leaf is a suffix prefix chain. In other words, each leaf can be used for identifying a specific suffix prefix chain. In fact, starting from a leaf and going backwards towards the root (with the exception of the root itself), we can reconstruct a specific suffix chain, in reverse order, by collecting the suffixes corresponding to the visited nodes. Figure 2 focuses on the subtree of the prefix tree showed in Figure 1 having root node a, and, using arrows in blue color, shows (on the left) how the suffix prefix chains from each leaf node are computed. We have that, the suffix prefix chains for the complete prefix tree showed in Figure 1 are: $C_1 = [a, aa, aaa]$, $C_2 = [a, aa, aabca]$, $C_3 = [a, abca]$, $C_4 = [b, bca]$, $C_5 = [ca, caabca]$, $C_6 = [dcaabca]$.

The construction of all C`-ranking` is performed by the `in-prefix-merge`, starting from the ranking list of a child of ϵ, and by stretching it by adding the ranking list of its children, as showed below. In general, the computation incrementally adds the ranking list $[s_{j+1}]$`-ranking` to the $[s_1, \ldots, s_j]$`-ranking`. Let $C = [s_1, \ldots, s_n]$ be a suffix prefix chain, *i.e.*, the sequence of suffixes we read on a path from s_1, child of the dummy node, to the leaf s_n. We compute the C`-ranking` as follows. Let us see how `in-prefix-merge(`$[s_1]$`-ranking`, $[s_2]$`-ranking)` works. As said, we start by adding the $[s_2]$`-ranking` to the $[s_1]$`-ranking`, *i.e.*, incrementally from the child of the dummy node to the leaves, where s_1 labels a child of ϵ and s_2 labels a child of s_1 (starting from the leftmost). Recall that s_1 is a prefix of s_2 and, by definition, $[s_1]$`-ranking` and $[s_2]$`-ranking` are

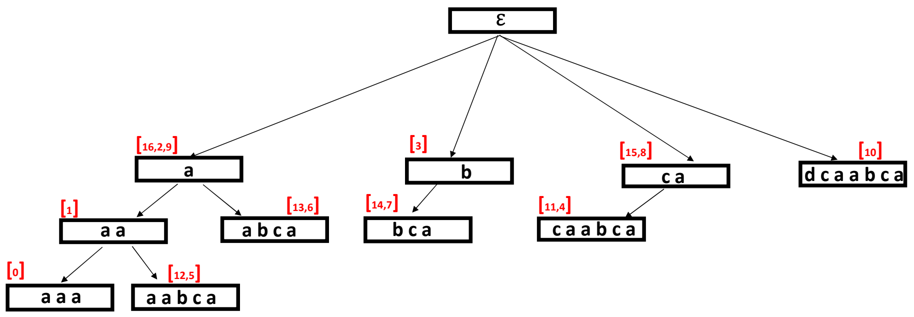

Fig. 1: The prefix tree for $\mathsf{ICFL}(S) = \langle aaa, b, caabca, dcaabca\rangle$. The ranking lists are reported in red.

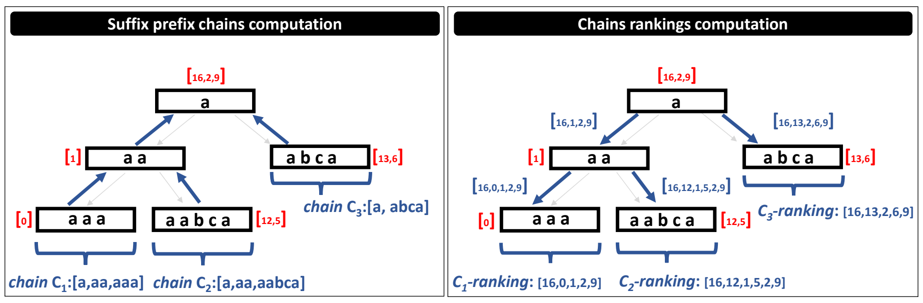

Fig. 2: Example of some suffix prefix chains (*left*) and their rankings (*right*).

sorted in such a way the global suffixes associated to the ranking lists are lexicographically sorted. Thus we describe how we can obtain the $[s_1, s_2]$-`ranking`. We remark that $[s_1, s_2]$ is not necessary a suffix prefix chain, since condition (ii) of Definition 2 may be not satisfied. However, with an abuse of notation, we may refer to a $[s_1, s_2]$-`ranking`, and in general to a C-`ranking`, even if $[s_1, s_2]$, and also C, does not satisfy condition (ii), in other words when C is simply a sequence of local suffixes which is increasing with respect to the prefix relation (condition (i)).

Assume that $[s_1]$-`ranking`$=[e_1, \ldots, e_n]$ and $[s_2]$-`ranking`$=[g_1, \ldots, g_h]$. Initially, we set $[s_1, s_2]$-`ranking` $= [s_1]$-`ranking` $= [e_1, \ldots, e_n]$. Then, we start by *comparing* g_1 with the elements e_i, where i goes from 1 to n, of $[s_1, s_2]$-`ranking`, until an integer e_i satisfying one of the cases below is found. Then we stop and g_1 is inserted immediately before e_i. After inserting g_1, we consider g_2 and we *compare* g_2 with $e_i, \ldots, e_n$, starting with e_i, and so on for all other elements of $[s_2]$-`ranking`.

Now, let us describe how the single *comparison* between g_1 and e_i, is made, where i goes from 1 to n. We have decided to describe this comparison here, outside the formal proof of Proposition 3, to avoid further complicating its reading.

We have to analyze three cases: (I) e_i, g_1 both occur in the last factor f_k, (II) e_i, g_1 both occur in the the same factor f_h, $h \neq k$, (III) e_i, g_1 occur in different factors.

(I) If both e_i and g_1 occur in the last factor f_k, then e_i should occur before g_1 in the final $[s_1, s_2]$-`ranking`. It is trivial to observe that $S_{e_i} < S_{g_1}$, since s_1, s_2 are both local suffixes of f_k and global suffixes of S. So, when e_i occurs in the last factor, g_1 is not inserted (and, later, g_1 will be compared with e_{i+1}).

(II) If both e_i and g_1 occur in the same factor f_l, with $f_l \neq f_k$, then g_1 occurs before e_i in f_l. Indeed, since $\overline{S}_{e_i}$ is prefix of $\overline{S}_{g_1}$, then g_1 precedes e_i. Thus, by Lemma 1, S_{g_1} precedes S_{e_i}, *i.e.*, g_1 should occur before e_i in the final $[s_1, s_2]$-`ranking`. In this case g_1 is inserted before e_i, *i.e.*, we now have $[s_1, s_2]$-`ranking`$=[e_1, \ldots, e_{i-1}, g_1, e_i, \ldots, e_n]$, and we compare g_2 with the elements of $[e_1, \ldots, e_{i-1}, g_1, e_i, \ldots, e_n]$, but starting from e_i.

(III) If e_i occurs in the factor f_s and g_1 occurs in the factor f_t and $t \neq s$, we can further distinguish three cases:

a) If $s = k$, then S_{e_i} precedes S_{g_1}. Indeed, being $\overline{S}_{e_i} = s_1$ prefix of $\overline{S}_{g_1} = s_2$ and so $|\overline{S}_{e_i}| < |\overline{S}_{g_1}|$, we have $S_{e_i} = \overline{S}_{e_i} < \overline{S}_{g_1} < S_{g_1}$ and so e_i precedes g_1, *i.e.*, g_1 is not inserted (and, later, g_1 will be compared with e_{i+1}).

b) If $t = k$, compute $p = \mathrm{LCP}(S_{e_i}, S_{g_1})$ and use the characters $\sigma, \sigma' \in \Sigma$, where $S_{e_i} = p\sigma z$, $S_{g_1} = p\sigma' z'$, to decide the ordering between S_{e_i} and S_{g_1}, being $\sigma \neq \sigma'$, and so whether to insert g_1 or not.

c) If $s \neq k \neq t$, we have two cases: if $e_i > g_1$, being $\overline{S}_{e_1}$ prefix of $\overline{S}_{g_1}$, we apply Lemma 1 by choosing $x = f_t \cdots f_s$. We observe that $\overline{S}_{e_i} = \overline{S}_{e_i,x}$ is a prefix of $\overline{S}_{g_1,x}$. So we have that g_1 precedes e_i, *i.e.*, g_1 is inserted before e_i. Otherwise, if $e_i < g_1$, Lemma 1 cannot be applied and so we proceed as in item b) for deciding whether inserting g_1 or not.

Recall that when computing LCP, the value of p is bounded by Proposition 1 (it is worthy of note that a more precise bound can be used thanks to other technical lemmas contained in [10]). The comparison described above for g_1 can be obviously done for each g_q, $q \in \{2, \ldots, h\}$, when $g_1, \ldots, g_{q-1}$ have already been inserted in the partial $[s_1, s_2]$-`ranking`. In addition, the same technique described above for obtaining the $[s_1, s_2]$-`ranking`, can be used for computing the $[s_1, s_2, s_3]$-`ranking` if s_3 is the leftmost child of s_2, and so on.

Example 2. Let us consider $[a]$-`ranking` $=[16, 2, 9]$ and $[aa]$-`ranking` $=[1]$. We compare 1 and 16 and we have case (IIIa), so 16 precedes 1. Then we compare 2 and 1, we have case (II) and so we insert 1, *i.e.*, $[a, aa]$-`ranking`$= [16, 1, 2, 9]$. Consider now $[a]$-`ranking` $=$ [16, 2, 9] and $[abca]$-`ranking` $= [13, 6]$. When comparing 16 and 13, which both belong to the last factor, we have that 16 comes before 13 (case (I)). Now, we compare 2 and 13 which belong to two different factors and we have case (IIIb) and so, by computing LCP, we have that 13 precedes 2. Now, we compare 2 and 6 which belong to the different factors (not the last factor) and so we have case (IIIc): we compute LCP and so we have that 2 precedes j. Finally, when we compare 9 and 6, we have case (II), and so 6 precedes 9. Finally $[a, abca]$-`ranking` $= [16, 13, 2, 6, 9]$. Figure 2 shows (on the right) how some such chain rankings are computed.

We can prove that the sequence of positions computed by the above described comparisons is the C-`ranking` for the suffix prefix chain C. The proof, reported in the Appendix, is done by induction on the levels of the prefix tree, precisely on the length of the path from a child of the dummy node to a leaf. As observed inside each case (I)-(III), the correctness of the computation is given by the Compatibility Property and, in the third case, also by using the LCP property for deciding the ordering between the two positions.

Proposition 3. *Let C be a suffix prefix chain. The sequence computed by the procedure* `in-prefix-merge` *described above is the C-*`ranking`*.*

3.3 Step3: merge the chain ranking lists to compute the Suffix Array

In this step, the ranking lists obtained in Step 2 for each suffix prefix chain are merged to build the final suffix array. Our strategy is described in the following.

Let $C_1, \ldots, C_m$ be the suffix prefix chains with their C_i-`ranking`, $i = 1, \ldots, m$. The idea is to organize $C_1, \ldots, C_m$ into `chain groups`, in this way: given the prefix tree described above, for each child node of the root, we build a specific `chain group` by collecting, from left to right, the suffix prefix chains corresponding to the leaf nodes of the subtree having such a node as root. For example, let us consider the tree in Figure 1. The root node has 4 child nodes: a, b, ca, and $dcaabca$. Thus, we build 4 `chain groups` as follows: the first one is obtained by considering the subtree having root node a and by collecting from left to right the suffix prefix chains corresponding to the leaf nodes aaa, $aabca$, and $abca$, i.e., $C_1 = [a, aa, aaa]$, $C_2 = [a, aa, aabca]$, and $C_3 = [a, abca]$; thus, the first `chain group` is $G_1 = [C_1, C_2, C_3]$; the other `chain groups` are built in a similar way, and so we have $G_2 = [C_4]$, $G_3 = [C_5]$, and $G_4 = [C_6]$.

Let $G = [C_1, \ldots, C_m]$ be a chain group. The goal is now to merge the C_1-`ranking`, C_2-`ranking`, and so on up to C_m-`ranking`, to build the G-`ranking`, *i.e.*, the sequence of all and only the positions of the C_i-`ranking`, $i = 1, \ldots, m$, without repetitions, and such that the sorting of the corresponding global suffixes is preserved. To this aim, consider $C_i = [s_{i_1}, \ldots, s_{i_g}]$ and $C_j = [s_{j_1}, \ldots, s_{j_t}]$, $i < j$, $i, j \in \{1, \ldots, m\}$. By construction, since they belong to the same subtree, they share a common path. So we have that $s_{i_1} = s_{j_1}$ and there exists q', $1 \leq q' \leq \min(g, t)$ such that $s_{i_q} = s_{j_q}$, for each $1 \leq q < q'$, and $s_{i_{q'}} < s_{j_{q'}}$. Thus, they share all the the local suffixes before q', *i.e.*, $s_{i_q} = s_{j_q}$, for each $1 \leq q < q'$. Clearly, when we merge their ranking lists, we find common elements, which correspond to the position of $s_{i_q} = s_{j_q}$.

Our strategy uses these common elements for computing the ranking lists, executing `common-prefix-merge`(C_1-`ranking`, ..., C_m-`ranking`), as follows. Specifically, we start with C_1-`ranking` and C_2-`ranking` and let $e_1, \ldots, e_t$ be the positions which are the common elements of C_1-`ranking` and C_2-`ranking`.

Now, we compute the *interleaving* of the elements of the C_1-`ranking` and of the C_2-`ranking` as follows. Start with an empty list. Then copy all the integers of C_1-`ranking` until e_1 is found, and stop here in C_1-`ranking`. Then add (copy) to this list all the integers of C_2-`ranking` until e_1 is found, stop here in C_2-`ranking` and then add e_1. Repeat the process on the integers of C_1-`ranking` (after e_1) until e_2 is found, and so on. It is worthy of note that since these chains have a common prefix path, they clearly share the elements contained in the ranking list of the lowest common ancestor node and these elements are already known since they have already been computed.

Observe that the computed list is the ranking list of the local suffixes of C_1 and C_2 in lexicographic order. Indeed, the common integers maintain the same relative order. Between two common integers, we interleave the integers which are not shared, by inserting the integers of the second list after the elements of the first list, and so on. Indeed, the elements which are not shared represent the positions of the local suffixes which are not contained in both suffix prefix chains. These local suffixes in the first chain are not proper prefixes of local suffixes in the second chain (they belong to different suffix prefix chains). However, by construction, we know that they are greater than the local suffixes of the second suffix prefix chain. So, by Lemma 1, we can prove that our interleaving is correct.

Proposition 4. *Let $G_1, \dots, G_s$ be the chain groups. The sequences $L_{G_1}, \dots, L_{G_s}$ computed by* `common-prefix-merge` *described above are G_r-*`ranking` *for each $r \in \{1, \dots, s\}$.*

Example 3. Consider $G_1 = [C_1, C_2, C_3]$, where $C_1 = [a, aa, aaa]$, $C_2 = [a, aa, aabca]$, and $C_3 = [a, abca]$. We start by merging $[a, aa, aaa]$-`ranking` $= [16, 0, 1, 2, 9]$ and $[a, aa, aabca]$-`ranking` $= [16, 12, 1, 5, 2, 9]$. We know that the common elements are $16, 1, 2, 9$ (the ranking list of the parent node aa, see Figure 2 on the right). So $[a,\ aa,\ aaa,\ aabca]$-`ranking`$= [16, 0, 12, 1, 5, 2, 9]$. Now, take $[a,\ abca]$-`ranking` $= [16, 13, 2, 6, 9]$. Then, `common-prefix-merge(`C_1`-ranking`, C_2`-ranking`, C_3`-ranking)`$=$ `common-prefix-merge(`[16, 0, 12, 1, 5, 2, 9], [16, 13, 2, 6, 9]`)` = [16, 0, 12, 1, 5, 13, 2, 6, 9]. See also Figure 3 in the Appendix.

Finally, let us consider the chain groups $G_1, \dots, G_s$, such that G_h is lexicographically smaller than G_{h+1}, $h = 1, \dots, s-1$. Let $X_h = G_h$-`ranking`, $h = 1, \dots, s$, constructed by using the `common-prefix-merge`. By construction, we have that $X_h = [e_{h_1}, \dots, e_{h_m}]$ is sorted in such a way, for each e_{h_i}, e_{h_j}, with $h_i < h_j$, $S_{e_{h_i}} < S_{e_{h_j}}$. The `concatenation-prefix-merge`$(X_1,\ \dots,\ X_s)$, simply incrementally concatenates $X_h = G_h$-`ranking`, for $h = 1, \dots, s$.

It is quite easy to see that the list computed by applying to $(X_1,\ \dots,\ X_s)$ the `concatenation-prefix-merge` is the final suffix array. Indeed, let $k_1, k_2 \in \{0, \dots, |[X]\text{-}\mathtt{ranking}| - 1\}$, $k_1 < k_2$, and denote by $q_1 = [X]$-`ranking`$[k_1]$ and $q_2 = [X]$-`ranking`$[k_2]$. We have that $S_{q_1} < S_{q_2}$, since, either q_1 and q_2 belong to the same $[G_i]$-`ranking` or $q_1 \in [G_i]$-`ranking`, $q_2 \in [G_j]$-`ranking`, $i < j$. In the first case, $S_{q_1} < S_{q_2}$, by Proposition 4. In the second case, being $i < j$, by construction $\overline{S}_{q_1} < \overline{S}_{q_2}$, and $\overline{S}_{q_1}$ is not a prefix of $\overline{S}_{q_2}$. This clearly implies that $S_{q_1} < S_{q_2}$. Thus, the following result holds.

Proposition 5. *The sequence computed by* `concatenation-prefix-merge` $(X_1, \dots, X_s)$ *is sequence of integers contained in the suffix array of* S.

4 On the implementation of the schema and future work

As already said in Section 1, the main goal of the paper is the investigation of a schema that uses the combinatorial properties of ICFL in the development of a new algorithmic strategy for sorting suffixes based on the idea of merging local suffixes as proposed in [18,19]. Indeed, in this paper we use novel properties of the canonical inverse Lyndon factorization that do not hold for the well known Lyndon factorization: precisely, the compatibility property (Lemma 1) and the bound on the longest common prefix of suffixes stated in Proposition 1. Clearly, the final objective of this investigation is understanding whether we can leverage these properties to speed up in practice the sorting of suffixes. The contributions given here are the result of a preliminary study in this direction mainly focused on developing a general schema than proving the algorithm with the best time complexity for sorting suffixes.

The complexity of an algorithm developed according to such a schema clearly mainly depends on the data structures used for building and maintaining the prefix tree structure proposed here. In particular, the final time complexity will be strongly related to the implementation of the proposed three steps. In the following we trace an analysis based on currently developed ideas.

An algorithm following the scheme proposed in Section 3, first computes $\mathsf{ICFL}(S)$. This step requires $\mathcal{O}(|S|)$ time. Then, it extracts the distinct local suffixes of $\mathsf{ICFL}(S)$: this step can be trivially done by processing S from right to left, and in $\mathcal{O}(|S|)$ time. After that, the performance is strongly related to the way we decide to manage the prefix tree to build the ranking lists of the local suffixes (Section 3.1), and to combine them in order to obtain the chain ranking lists (Section 3.2) and groups (Section 3.3). For example, a crucial aspect concerns the construction of the tree. In fact, we can imagine two main scenarios: *(i)* the extraction of the distinct local suffixes, with relative lexicographic ordering, and the computation of the rankings (`Step1` and `Step2`) takes place before the construction of the tree, and only subsequently the tree is used for `Step3`; *(ii)* the tree is built during the extraction of the distinct local suffixes and the computation of the rankings. Below, we discuss these scenarios in more details.

As for the first scenario, after the computation of $\mathsf{ICFL}(S)$ and the extraction of the distinct local suffixes, such suffixes are sorted alphabetically. We remark that for this task we can choose any known algorithm existing in literature, as suggested in [19]. Notice, for example, that we have at most $|S|$ local suffixes (on a finite alphabet Σ) of length at most $\max\{|f_i| \mid 1 \leq i < k\}$, and so we could use an adaptation of the *radix sort*. However, in general we say that such task requires $\mathcal{O}(\tau)$ time, where τ is the time taken to execute the chosen sorting technique. In order to built the ranking lists of the distinct local suffixes of $\mathsf{ICFL}(S) = \langle f_1, \dots, f_k \rangle$, we can maintain them in a dictionary with keys lexicographically sorted. Each local suffix maintains the reference to the factor which contains it, so that we can recover this information in constant time. Finally, the ranking lists are filled simply by running S from f_{k-1} to f_1 and finally f_k. Also this operation requires $\mathcal{O}(|S|)$ time, since each insertion of an element in a ranking list is done at the beginning of the list, *i.e.*, in constant time. At this point, the construction of the tree can be done simply by scrolling through the sorted suffixes and creating a node of the tree for each one. The advantage of building the prefix tree starting with alphabetically ordered strings is that this can be done in linear time in the number of strings ($\mathcal{O}(|S|)$). This is actually the idea used in a prototype implementation of our strategy, that can be freely downloaded[3].

As for the second scenario, used in an ongoing implementation, the main difference is that we avoid the dictionary and the tree is built when the local distinct suffixes are extracted. To optimize the construction, we can extract the suffixes in order of increasing length, and whenever a new suffix (a new node) must be inserted into the tree, the nodes which represent shorter suffixes have certainly already been inserted. Thus we don't have to modify the structure of the tree in the higher levels. However, having identified the parent node to hook the new node to, we must ensure that it is positioned in alphabetical order (with respect to the suffix) among the child nodes of the parent node itself. This can be done by applying a binary search, since each time the insertion takes place within an already sorted list, in $\mathcal{O}(|S| \times log|S|)$ time in the worst case. As for the first scenario, the overall update of the ranking lists can be performed in $\mathcal{O}(|S|)$. For both the scenarios, the suffix prefix chains and their ranking lists can be computed by visiting the tree to reach all the leaf nodes, and by merging at each step two lists L_1 and L_2. According to Lemma 2, by construction they contain positions of suffixes already lexicographically ordered. This requires $\mathcal{O}(|L_1| + |L_2|)$ comparisons, at each step. Recall that each comparison can be done either in $\mathcal{O}(1)$ or in $\mathcal{O}(\mathcal{M})$, when the LCP property is used. Finally, `Step3` is performed using the tree by executing the `common-prefix-merge`, that can take advantage of the ranking lists already computed in each node, and the `concatenation-prefix-merge`, that is a trivial concatenation of lists.

As future works, we will investigate other merge operations in order to avoid some comparisons. Moreover, it is possible to give also an external memory implementation of our algorithm, and a parallel strategy for analyzing the prefix tree.

References

1. Bahne, J., Bertram, N., Böcker, M., Bode, J., Fischer, J., Foot, H., Grieskamp, F., Kurpicz, F., Löbel, M., Magiera, O., Pink, R., Piper, D., Poeplau, C.: Sacabench: Benchmarking suffix array construction. In: SPIRE. pp. 407–416 (2019)
2. Baier, U.: Linear-time Suffix Sorting - A New Approach for Suffix Array Construction. In: CPM 2016. vol. 54, pp. 23:1–23:12
3. Bannai, H., Tomohiro, I., Inenaga, S., Nakashima, Y., Takeda, M., Tsuruta, K.: A new characterization of maximal repetitions by Lyndon trees. In: SODA 2015. pp. 562–571

[3] `https://github.com/rzaccagnino/icfl2sa`

4. Bertram, N., Ellert, J., Fischer, J.: Lyndon Words Accelerate Suffix Sorting. In: ESA 2021. vol. 204, pp. 15:1–15:13
5. Bonizzoni, P., Costantini, M., De Felice, C., Petescia, A., Pirola, Y., Previtali, M., Rizzi, R., Stoye, J., Zaccagnino, R., Zizza, R.: Numeric Lyndon-based feature embedding of sequencing reads for machine learning approaches. Inf. Sci. **607**, 458–476 (2022)
6. Bonizzoni, P., De Felice, C., Petescia, A., Pirola, Y., Rizzi, R., Stoye, J., Zaccagnino, R., Zizza, R.: Can we replace reads by numeric signatures? Lyndon fingerprints as representations of sequencing reads for machine learning. In: AlCoB 2021. Lect. Not. in Comp. Sci., vol. 12715, pp. 16–28. Springer (2021)
7. Bonizzoni, P., De Felice, C., Pirola, Y., Rizzi, R., Zaccagnino, R., Zizza, R.: Can formal languages help pangenomics to represent and analyze multiple genomes? In: Developments in Language Theory - 26th International Conference, DLT 2022. Lecture Notes in Computer Science, vol. 13257, pp. 3–12. Springer (2022)
8. Bonizzoni, P., De Felice, C., Zaccagnino, R., Zizza, R.: Inverse Lyndon words and inverse Lyndon factorizations of words. Adv. Appl. Math. **101**, 281–319 (2018)
9. Bonizzoni, P., De Felice, C., Zaccagnino, R., Zizza, R.: Lyndon words versus inverse Lyndon words: Queries on suffixes and bordered words. In: LATA 2020. Lecture Notes in Computer Science, vol. 12038, pp. 385–396 (2020)
10. Bonizzoni, P., De Felice, C., Zaccagnino, R., Zizza, R.: On the longest common prefix of suffixes in an inverse Lyndon factorization and other properties. Theor. Comput. Sci. **862**, 24–41 (2021)
11. Bonizzoni, P., Petescia, A., Pirola, Y., Rizzi, R., Zaccagnino, R., Zizza, R.: Kfinger: Capturing overlaps between long reads by using Lyndon fingerprints. In: IWBBIO Conference. Lect.Notes in Comp. Sci., vol. 13347, pp. 436–449. Springer (2022)
12. Chen, K.T., Fox, R.H., Lyndon, R.C.: Free Differential Calculus, IV. The quotient groups of the Lower Central Series. Ann. Math. **68**, 81–95 (1958)
13. Duval, J.: Factorizing words over an ordered alphabet. J. Algorithms **4**(4), 363–381 (1983)
14. Fischer, J., Kurpicz, F.: Dismantling divsufsort. In: Holub, J., Zdárek, J. (eds.) PSC 2017. pp. 62–76
15. Goto, K.: Optimal time and space construction of suffix arrays and LCP arrays for integer alphabets. In: Holub, J., Zdárek, J. (eds.) PSC2019. pp. 111–125
16. Lothaire, M.: Algebraic Combinatorics on Words, Encyclopedia Math. Appl., vol. 90. Cambridge University Press (1997)
17. Lyndon, R.: On Burnside problem i. Trans. Amer. Math. Soc. **77**, 202–215 (1954)
18. Mantaci, S., Restivo, A., Rosone, G., Sciortino, M.: Sorting Suffixes of a Text via its Lyndon Factorization. In: Proceedings of the Prague Stringology Conference 2013, Prague, Czech Republic, September 2-4, 2013. pp. 119–127 (2013)
19. Mantaci, S., Restivo, A., Rosone, G., Sciortino, M.: Suffix array and Lyndon factorization of a text. J. Discrete Algorithms **28**, 2–8 (2014)

Appendix

This Appendix provides the readers with the proofs of Propositions 2 and 3 not included in the main text. In addition, we also give two supplementary figures, one regards the Step 3 and the second one presents an overview of the proposed strategy.

Proof of Proposition 2. Let $L_s = [e_1, \ldots, e_m]$ be the list obtained by Step1 for the local suffix s. By construction, $s = \overline{S}_{e_r}$, for each $r \in \{1, \ldots, m\}$ and so condition *(i)* of Definition 1 is satisfied. We prove that condition *(ii)* is satisfied, by showing that for each $e_i, e_j \in L_s$ such that e_i occurs before e_j in L_s, then $S_{e_i} < S_{e_j}$.

Recall that Step1 analyzes the factors of $\mathsf{ICFL}(S)$ from f_{k-1} to f_1 and lastly f_k. In addition, we have $f_{k-1} \gg f_{k-2} \gg \ldots \gg f_1$ by definition. Thus, if s is a suffix of the factors f_h and $f_{h'}$, with $h < h'$ and $h' \neq k$, by construction the occurrence of s in $f_{h'}$ is considered before the occurrence of s in f_h. In other words, Step1 first adds at the beginning of the list the occurrence of s in $f_{h'}$, say e_j, and, later, the occurrence of s in f_h, say e_i. Indeed, this is the reason why we found e_i before e_j in L_s (since by hypothesis we have that e_i occurs before e_j in L_s). But, since $sf_{h+1} \cdots f_k < sf_{h'+1} \cdots f_k$, being $f_{h+1} \ll f_{h'+1}$, we have that $S_{e_i} = sf_{h+1} \cdots f_k < sf_{h'+1} \cdots f_k = S_{e_j}$, concluding the proof: the construction performed by Step1 preserves the ordering of the corresponding global suffixes starting with s.

Now, let us suppose that $h' = k$, *i.e.*, s occurs as a local suffix of the last factor, so s is also a global suffix: the occurrence of s in $f_{h'} = f_k$ is inserted at the beginning of the ranking list as the last element. Thus, if $s \in Suff(f_k)$, then the occurrence of s in f_k will be the first element of L_s, *i.e.*, e_1. Hence, we have that $S_{e_1} = \overline{S}_{e_1} = s$ and this is correct, since $S_{e_1} < S_{e_r}$, for each $r \in \{2, \ldots, m\}$. Again, Step1 preserves the ordering of the corresponding global suffixes starting with s. ■

Proof of Proposition 3. We denote $s <_p s'$ if s is a prefix of s'. Let C be the suffix prefix chain $C = [s_1, \ldots, s_n]$. By definition, $s_1 <_p s_2 <_p \ldots <_p s_n$ and let $[e_1, \ldots, e_t]$ the sequence L_C computed by `in-prefix-merge`. We prove that $L_C = C$-`ranking` by induction on n. For simplicity, in this proof we name "root" the node of the prefix tree containing s_1. By construction, n is exactly the length of the path from the root s_1 to s_n, *i.e.*, the levels in the tree, minus 1, if we consider that the root s_1 is at level 0.

Base Step. If $n = 1$, then the $C = [s_1]$-`ranking` is exactly its ranking list constructed in the Step 1 and so it is the $C = [s_1]$-`ranking`, by Proposition 2.

Induction Step. Suppose that for $C' = [s_1, \ldots, s_{n-1}]$. By induction hypothesis, the sequence $L_{C'}$ constructed by the `in-prefix-merge` is the C'-`ranking`. Let us show that when we apply `in-prefix-merge`(C'-`ranking`, $[s_n]$-`ranking`), we obtain C-`ranking`=$[s_1, \ldots, s_n]$-`ranking`. Let $[s_n]$-`ranking`= $[g_1, \ldots, g_r]$. Also this step is proved by induction, on r.

Suppose that $r = 1$, *i.e.*, there is only one occurrence in g_1 of s_n as a suffix of a local suffix and analyze the three cases $(I) - (III)$, when comparing g_1 and e_1 (recall that we start from e_1; a note about this is reported in the last section).

If g_1 and e_1 were in the last factor, case (I) implies that e_1 precedes g_1 since $S_{e_1} < S_{g_1} = s_n$, being $s_i <_p s_n$, for each $i \in \{1, \ldots, n-1\}$. In this case g_1 is not inserted in the list now. In fact, since each e_i is a starting position of a local suffix which is a prefix of g_1, if g_1 is in the last factor, it is inserted after all the positions e_i belonging to the last factor. In fact we will search for the first element e_i which does not belong to the last factor (and so we fall in case (III)).

If g_1 and e_1 were in the same factor, which is not the last factor, then by case (II) we insert g_1 in the new list (and so before e_1). Indeed, being g_1 and e_1 in the same factor, but being s_n longer than each s_i, obviously g_1 is smaller than e_1. Thus, by Lemma 1, S_{g_1} precedes S_{e_1}, *i.e.*, g_1 comes before e_1 in the new list (observe that in Lemma 1 we choose the substring x of the statement as the factor of ICFL containing both ranks).

Suppose that g_1 and e_1 occur in different factors, say f_t and f_s, respectively.

If $s = k$, clearly S_{e_1} precedes S_{g_1}. Indeed, $s_i <_p s_n$, for each $i \in \{1, \ldots, n-1\}$, and $|\overline{S}_{e_1}| < |\overline{S}_{g_1}|$. So $S_{e_1} = \overline{S}_{e_1} < \overline{S}_{g_1} < S_{g_1}$. Correctly, `in-prefix-merge` inserts e_1 before g_1 (case IIIa).

If $t = k$, we need to compute $p = \mathrm{LCP}(S_{e_1}, S_{g_1})$ and use the characters $\sigma, \sigma' \in \Sigma$, where $S_{e_1} = p\sigma z$, $S_{g_1} = p\sigma' z'$, to decide the ordering between e_1 and g_1, being $\sigma \neq \sigma'$ and so to decide whether to insert g_1 or not (case IIIb).

Finally, let $s \neq k \neq t$. We can consider two cases. If $e_1 > g_1$, since e_1 is the position of a local suffix s_i and $s_i <_p s_n$, for each $i \in \{1, \ldots, n-1\}$, we can apply Lemma 1, by choosing $x = f_t \cdots f_s$ and by observing that $\overline{S}_{e_1} = \overline{S}_{e_1,x}$ is a prefix of $\overline{S}_{g_1,x}$. So we have that g_1 precedes e_1 and indeed g_1 is inserted before e_1 (case (IIIc)). Otherwise, if $e_1 < g_1$, Lemma 1 cannot be applied and we proceed by computing $\mathrm{LCP}(S_{g_1}, S_{e_1})$ for deciding whether to insert g_1 or not.

Suppose now that $r > 1$. Denote by $[e_1, \ldots, e_{t'}]$ the ranking list (it is the ranking list by induction hypothesis) containing both the positions of $s_1, \ldots, s_{n-1}$ and of $g_1, \ldots, g_{r-1}$. Recall that, by definition, S_{g_r} is the greatest suffix starting with s_n (and, clearly, such that this occurrence of s_n is a local suffix) and $S_{g_j} < S_{g_r}$, for each $j \in \{1, \ldots, r-1\}$.

Suppose that g_{r-1} was inserted due to the comparison with $e_v, v \in \{1, \ldots, t'\}$, *i.e.*, case (II) or (III) happened. So the occurrence of s_n at position g_{r-1} was inserted before the occurrence of S_{e_v}. Now, consider again the three cases. If g_r and e_v belong to the last factor, *i.e.*, case (I) occurs, and so g_r will be inserted later, as soon we find a rank in $\{e_{v+1}, \ldots, e_{t'}\}$ which does not belong to the last factor. If this position does not exist, then we insert g_r at the end, and, in this case we have that $S_{e_{v+1}} < \cdots < S_{e_{t'}} < S_{e_g}$ is true and the proof ended.

Otherwise, suppose that there exists a position $e_q \in \{e_v, \ldots, e_{t'}\}$ which does not belong to the last factor, and so this is a special case (II). Thus we add g_s before e_q, and this is correct since we know that $S_{e_v} < \cdots < S_{e_{q-1}} < S_{g_r}$ and $S_{g_r} < S_{e_q}$. So we have $S_{e_1} < \cdots < S_{e_v} < \cdots < S_{e_{q-1}} < S_{g_r} < S_{e_q} < \cdots < S_{t'}$, and the proof ended. A similar proof can be done when both g_r and e_q belong to the same factor (different from the last factor) and in case (III). ∎

The figure below visualizes the construction, on the string $S = aaabcaabcadcaabca$, of groups and their rankings (Step3), as partially described in Example 3. For each node (except for the leaf nodes), the result of the interleaving of the children nodes is a list in which the blue items are the items of the ranking of the node itself (common elements) while the orange items are the those one taken by the children nodes (from left to right).

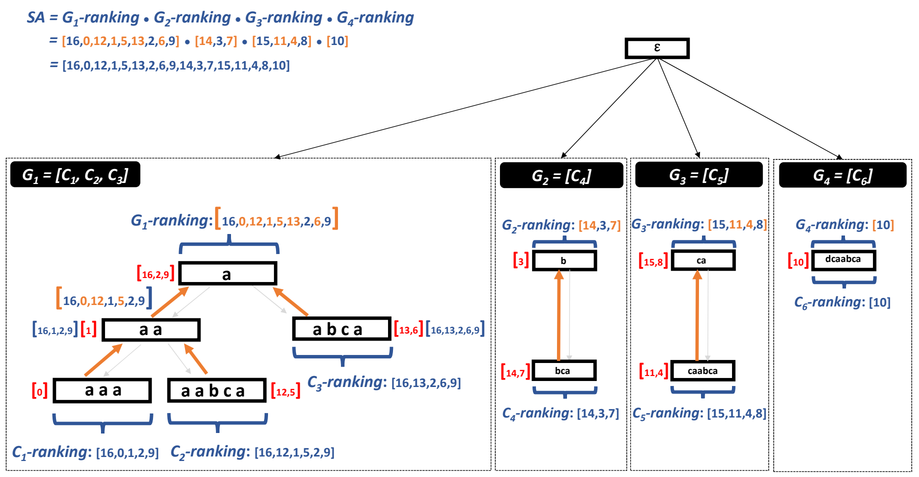

Fig. 3

Finally, next figure is an overview of the proposed strategy on the string $S = aaabcaabcadcaabca$.

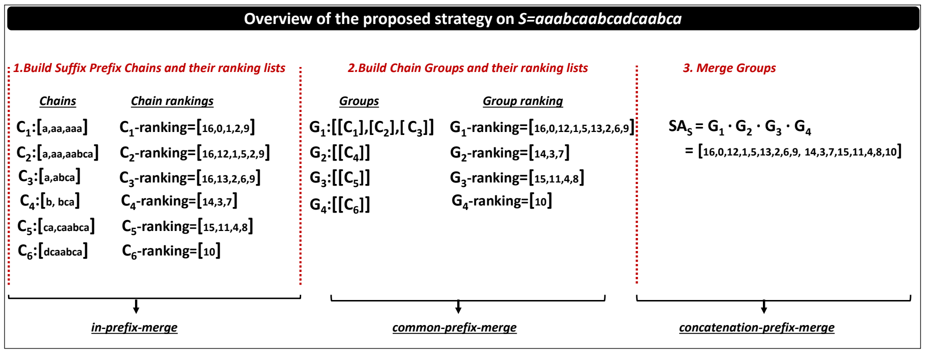

Fig. 4