On the Optimisation of the GSACA Suffix Array Construction Algorithm

Jannik Olbrich(✉), Enno Ohlebusch, and Thomas Büchler

University of Ulm, 89081 Ulm, Germany
{jannik.olbrich,enno.ohlebusch,thomas.buechler}@uni-ulm.de
https://www.uni-ulm.de/in/theo

Abstract. The suffix array is arguably one of the most important data structures in sequence analysis and consequently there is a multitude of suffix sorting algorithms. However, to this date the `GSACA` algorithm introduced in 2015 is the only known non-recursive linear-time suffix array construction algorithm (SACA). Despite its interesting theoretical properties, there has been little effort in improving the algorithm's sub-par real-world performance. There is a super-linear algorithm `DSH` which relies on the same sorting principle and is faster than `DivSufSort`, the fastest SACA for over a decade. This paper is concerned with analysing the sorting principle used in `GSACA` and `DSH` and exploiting its properties in order to give an optimised linear-time algorithm. Our algorithm is not only significantly faster than `GSACA` but also outperforms `DivSufSort` and `DSH`.

Keywords: Suffix array · Suffix sorting · String algorithms

1 Introduction

The *suffix array* contains the indices of all suffixes of a string arranged in lexicographical order. It is arguably one of the most important data structures in *stringology*, the topic of algorithms on strings and sequences. It was introduced in 1990 by Manber and Myers for on-line string searches [9] and has since been adopted in a wide area of applications including text indexing and compression [12]. Although the suffix array is conceptually very simple, constructing it efficiently is not a trivial task.

When n is the length of the input text, the suffix array can be constructed in $\mathcal{O}(n)$ time and $\mathcal{O}(1)$ additional words of working space when the alphabet is linearly-sortable (i.e. the symbols in the string can be sorted in $\mathcal{O}(n)$ time) [7,8,10]. However, algorithms with these bounds are not always the fastest in practice. For instance, `DivSufSort` has been the fastest SACA for over a decade although having super-linear worst-case time complexity [3,5]. To the best of our knowledge, the currently fastest suffix sorter is `libsais`, which appeared as source code in February 2021 on Github[1] and has not been subject to peer

[1] https://github.com/IlyaGrebnov/libsais, last accessed: August 22, 2022.

review in any academic context. The author claims that `libsais` is an improved implementation of the SA-IS algorithm and hence has linear time complexity [11].

The only non-recursive linear-time suffix sorting algorithm `GSACA` was introduced in 2015 by Baier and is not competitive, neither in terms of speed nor in the amount of memory consumed [1,2]. Despite the new algorithm's entirely novel approach and interesting theoretical properties [6], there has been little effort in optimising it. In 2021, Bertram et al. [3] provided a faster SACA `DSH` using the same sorting principle as `GSACA`. Their algorithm beats `DivSufSort` in terms of speed, but also has super-linear time complexity.

Our Contributions. We provide a linear-time SACA that relies on the same *grouping* principle that is employed by `DSH` and `GSACA`, but is faster than both. This is done by exploiting certain properties of Lyndon words that are not used in the other algorithms. As a result, our algorithm is more than 11% faster than `DSH` on real-world texts and at least 46% faster than Baier's `GSACA` implementation. Although our algorithm is not on par with `libsais` on real-world data, it significantly improves Baier's sorting principle and positively answers the question whether the precomputed Lyndon array can be used to accelerate `GSACA` (posed in [4]).

The rest of this paper is structured as follows: Sect. 2 introduces the definitions and notations used throughout this paper. In Sect. 3, the grouping principle is investigated and a description of our algorithm is provided. Finally, in Sect. 4 our algorithm is evaluated experimentally and compared to other relevant SACAs.

This is an abridged version of a longer paper available on arXiv [13].

2 Preliminaries

For $i, j \in \mathbb{N}_0$ we denote the set $\{k \in \mathbb{N}_0 : i \le k \le j\}$ by the interval notations $[i \mathinner{..} j] = [i \mathinner{..} j+1) = (i-1 \mathinner{..} j] = (i-1 \mathinner{..} j+1)$. For an array A we analogously denote the *subarray* from i to j by $A[i \mathinner{..} j] = A[i \mathinner{..} j+1) = A(i-1 \mathinner{..} j] = A(i-1 \mathinner{..} j+1) = A[i]A[i+1] \dots A[j]$. We use zero-based indexing, i.e. the first entry of the array A is $A[0]$. A *string* S of *length* n over an *alphabet* Σ is a sequence of n characters from Σ. We denote the length n of S by $|S|$ and the i'th symbol of S by $S[i-1]$, i.e. strings are zero-indexed. Analogous to arrays we denote the *substring* from i to j by $S[i \mathinner{..} j] = S[i \mathinner{..} j+1) = S(i-1 \mathinner{..} j] = S(i-1 \mathinner{..} j+1) = S[i]S[i+1] \dots S[j]$. For $j > i$ we let $S[i \mathinner{..} j]$ be the *empty string* ε. The *suffix* i of a string S of length n is the substring $S[i \mathinner{..} n)$ and is denoted by S_i. Similarly, the substring $S[0 \mathinner{..} i]$ is a *prefix* of S. A suffix (prefix) is *proper* if $i > 0$ $(i+1 < n)$. For two strings u and v and an integer $k \ge 0$ we let uv be the concatenation of u and v and denote the k-times concatenation of u by u^k. We assume totally ordered alphabets. This induces a total order on strings. Specifically, we say a string S of length n is *lexicographically smaller*

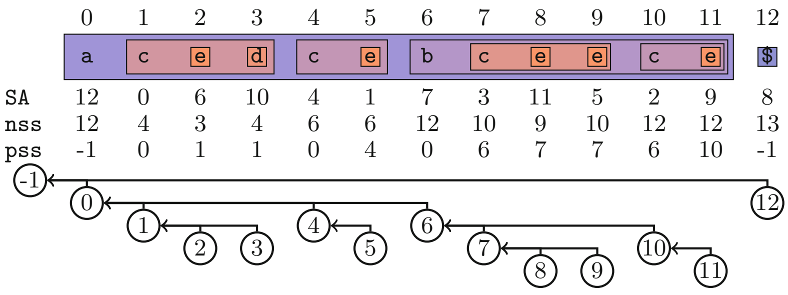

0 1 2 3 4 5 6 7 8 9 10 11 12
a c e d c e b c e e c e $
SA 12 0 6 10 4 1 7 3 11 5 2 9 8
nss 12 4 3 4 6 6 12 10 9 10 12 12 13
pss -1 0 1 1 0 4 0 6 7 7 6 10 -1
-1
0
1
2
3
4
5
6
7
8
9
10
11
12

Fig. 1. Shown are the Lyndon prefixes of all suffixes of $S =$ `acedcebceece$` and the corresponding suffix array, `nss`-array, `pss`-array and `pss`-tree. Each box indicates a Lyndon prefix. For instance, the Lyndon prefix of $S_7 =$ `ceece$` is $\mathcal{L}_7 =$ `cee`. Note that $\mathcal{L}_i$ is exactly $S[i]$ concatenated with the Lyndon prefixes of i's children in the `pss`-tree (see Lemma 4), e.g. $\mathcal{L}_6 = S[6]\mathcal{L}_7\mathcal{L}_{10} =$ `bceece`.

than another string S' of length m if and only if there is some $\ell \le \min\{n, m\}$ such that $S[0..\ell) = S'[0..\ell)$ and either $n = \ell < m$ or $S[\ell] < S'[\ell]$. If S is lexicographically smaller than S' we write $S <_{lex} S'$.

A non-empty string S is a *Lyndon word* if and only if S is lexicographically smaller than all its proper suffixes [14]. The *Lyndon prefix* of S is the longest prefix of S that is a Lyndon word. We let $\mathcal{L}_i$ denote the Lyndon prefix of S_i.

In the remainder of this paper, we assume an arbitrary but fixed string S of length $n > 1$ over a totally ordered alphabet Σ with $|\Sigma| \in \mathcal{O}(n)$. Furthermore, we assume w.l.o.g. that S is *null-terminated*, that is $S[n-1] = \$$ and $S[i] > \$$ for all $i \in [0..n-1)$.

The *suffix array* `SA` of S is an array of length n that contains the indices of the suffixes of S in increasing lexicographical order. That is, `SA` forms a permutation of $[0..n)$ and $S_{\mathtt{SA}[0]} <_{lex} S_{\mathtt{SA}[1]} <_{lex} \dots <_{lex} S_{\mathtt{SA}[n-1]}$.

Definition 1 (pss-tree [4]). *Let* `pss` *be the array such that* $\mathtt{pss}[i]$ *is the index of the previous smaller suffix for each* $i \in [0..n)$ *(or -1 if none exists). Formally,* $\mathtt{pss}[i] := \max(\{j \in [0..i) : S_j <_{lex} S_i\} \cup \{-1\})$. *Note that* `pss` *forms a tree with -1 as the root, in which each* $i \in [-1..n)$ *is represented by a node and* $\mathtt{pss}[i]$ *is the parent of node i. We call this tree the* `pss`-tree*. Further, we impose an order on the nodes that corresponds to the order of the indices represented by the nodes. In particular, if* $c_1 < c_2 < \dots < c_k$ *are the children of* i *(i.e.* $\mathtt{pss}[c_1] = \dots = \mathtt{pss}[c_k] = i$*), we say* c_k *is the* last child *of* i.

Analogous to $\mathtt{pss}[i]$, we define $\mathtt{nss}[i] := \min\{j \in (i..n] : S_j <_{lex} S_i\}$ as the next smaller suffix of i. Note that $S_n = \varepsilon$ is smaller than any non-empty suffix of S, hence `nss` is well-defined.

In the rest of this paper, we use $S =$ `acedcebceece$` as our running example. Figure 1 shows its Lyndon prefixes and the corresponding `pss`-tree.

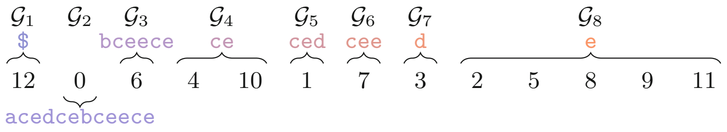


Fig. 2. A Lyndon grouping of `acedcebceece$` with group contexts.

Definition 2. *Let $\mathcal{P}_i$ be the set of suffixes with i as next smaller suffix, that is*

$$\mathcal{P}_i = \{j \in [0 \mathinner{..} i) : \texttt{nss}[j] = i\}$$

For instance, in the example we have $\mathcal{P}_4 = \{1, 3\}$ because $\texttt{nss}[1] = \texttt{nss}[3] = 4$.

3 GSACA

We start by giving a high level description of the sorting principle based on grouping by Baier [1,2]. Very basically, the suffixes are first assigned to lexicographically ordered groups, which are then refined until the suffix array emerges. The algorithm consists of the following steps.

- *Initialisation:* Group the suffixes according to their first character.
- *Phase I:* Refine the groups until the elements in each group have the same Lyndon prefix.
- *Phase II:* Sort elements within groups lexicographically.

Definition 3 (Suffix Grouping, adapted from [3]). *Let S be a string of length n and $\mathtt{SA}$ the corresponding suffix array. A* group *$\mathcal{G}$ with* group context *α is a tuple $\langle g_s, g_e, |\alpha|\rangle$ with* group start *$g_s \in [0 \mathinner{..} n)$ and* group end *$g_e \in [g_s \mathinner{..} n)$ such that the following properties hold:*

1. *All suffixes in $\mathtt{SA}[g_s \mathinner{..} g_e]$ share the prefix α, i.e. for all $i \in \mathtt{SA}[g_s \mathinner{..} g_e]$ it holds $S_i = \alpha S_{i+|\alpha|}$.*
2. *α is a Lyndon word.*

We say i is in *$\mathcal{G}$ or i* is an element of *$\mathcal{G}$ and write $i \in \mathcal{G}$ if and only if $i \in \mathtt{SA}[g_s \mathinner{..} g_e]$. A* suffix grouping *for S is a set of groups $\mathcal{G}_1, \ldots, \mathcal{G}_m$, where the groups are pairwise disjoint and cover the entire suffix array. Formally, if $\mathcal{G}_i = \langle g_{s,i}, g_{e,i}, |\alpha_i|\rangle$ for all i, then $g_{s,1} = 0, g_{e,m} = n-1$ and $g_{s,j} = 1 + g_{e,j-1}$ for all $j \in [2 \mathinner{..} m]$. For $i, j \in [1 \mathinner{..} m]$, $\mathcal{G}_i$ is a* lower (higher) *group than $\mathcal{G}_j$ if and only if $i < j$ ($i > j$). If all elements in a group $\mathcal{G}$ have α as their Lyndon prefix then $\mathcal{G}$ is a* Lyndon group*. If $\mathcal{G}$ is not a Lyndon group, it is called* preliminary*. Furthermore, a suffix grouping is* Lyndon *if all its groups are Lyndon groups, and* preliminary *otherwise.*

With these notions, a suffix grouping is created in the initialisation, which is then refined in Phase I until it is Lyndon, and further refined in Phase II until the suffix array emerges. Figure 2 shows a Lyndon grouping of our running example.

In Subsects. 3.1 and 3.2 we explain Phases II and I, respectively, of our suffix array construction algorithm. Phase II is described first because it is much simpler.

```
$A[0] \leftarrow n - 1$;
for $i = 0 \rightarrow n - 1$ do
    for $j \in \mathcal{P}_{A[i]}$ do
        Let $k$ be the start of the group containing $j$;
        remove $j$ from its current group and put it in a new group $\langle k, k, |\mathcal{L}_j|\rangle$ immediately
          preceding $j$'s old group;
        $A[k] \leftarrow j$;
    end
end
```

Algorithm 1: Phase II of `GSACA` [1,2]

3.1 Phase II

In Phase II we need to refine the Lyndon grouping obtained in Phase I into the suffix array. Let $\mathcal{G}$ be a Lyndon group with context α and let $i, j \in \mathcal{G}$. Since $S_i = \alpha S_{i+|\alpha|}$ and $S_j = \alpha S_{j+|\alpha|}$, we have $S_i <_{lex} S_j$ if and only if $S_{i+|\alpha|} <_{lex} S_{j+|\alpha|}$. Hence, in order to find the lexicographically smallest suffix in $\mathcal{G}$, it suffices to find the lexicographically smallest suffix p in $\{i + |\alpha| : i \in \mathcal{G}\}$. Note that removing $p - |\alpha|$ from $\mathcal{G}$ and inserting it into a new group immediately preceding $\mathcal{G}$ yields a valid Lyndon grouping. We can repeat this process until each element in $\mathcal{G}$ is in its own singleton group. As $\mathcal{G}$ is Lyndon, we have $S_{k+|\alpha|} <_{lex} S_k$ for each $k \in \mathcal{G}$. Therefore, if all groups lower than $\mathcal{G}$ are singletons, p can be determined by a simple scan over $\mathcal{G}$ (by determining which member of $\{i + |\alpha| : i \in \mathcal{G}\}$ is in the lowest group). Consider for instance $\mathcal{G}_4 = \langle 3, 4, |\mathtt{ce}|\rangle$ from Fig. 2. We consider $4 + |\mathtt{ce}| = 6$ and $10 + |\mathtt{ce}| = 12$. Among them, 12 belongs to the lowest group, hence S_{10} is lexicographically smaller than S_4. Thus, we know $\mathtt{SA}[3] = 10$ and remove 10 from $\mathcal{G}_4$ and repeat the process with the emerging group $\mathcal{G}'_4 = \langle 4, 4, |\mathtt{ce}|\rangle$. As $\mathcal{G}'_4$ only contains 4 we know $\mathtt{SA}[4] = 4$.

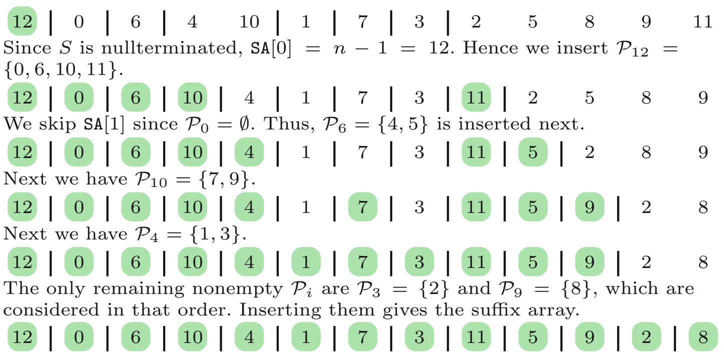
If the groups are refined from lower to higher as just described, each time a group $\mathcal{G}$ is processed, all groups lower than $\mathcal{G}$ are singletons. However, sorting groups in such a way leads to a superlinear time complexity. Bertram et al. [3] provide a fast-in-practice $\mathcal{O}(n \log n)$ algorithm for this, broadly following the described approach.

In order to get a linear time complexity, we turn this approach on its head like Baier does [1,2]: Instead of repeatedly finding the next smaller suffix in a group, we consider the suffixes in lexicographically increasing order and for each encountered suffix i, we move all suffixes that have i as the next smaller suffix (i.e. those in $\mathcal{P}_i$) to new singleton groups immediately preceding their respective old groups. Corollary 1 implies that this procedure is well-defined.

Lemma 1. *For any $j, j' \in \mathcal{P}_i$ we have $\mathcal{L}_j \neq \mathcal{L}_{j'}$ if and only if $j \neq j'$.*

Corollary 1. *In a Lyndon grouping, the elements of $\mathcal{P}_i$ are in different groups.*

Accordingly, Algorithm 1 correctly computes the suffix array from a Lyndon grouping. A formal proof of correctness is given in [1,2]. Figure 3 shows Algorithm 1 applied to our running example.



$\boxed{12} \mid 0 \mid 6 \mid 4 \quad 10 \mid 1 \mid 7 \mid 3 \mid 2 \quad 5 \quad 8 \quad 9 \quad 11$

Since S is nullterminated, $\texttt{SA}[0] = n - 1 = 12$. Hence we insert $\mathcal{P}_{12} = \{0, 6, 10, 11\}$.

$\boxed{12} \mid \boxed{0} \mid \boxed{6} \mid \boxed{10} \mid 4 \mid 1 \mid 7 \mid 3 \mid \boxed{11} \mid 2 \quad 5 \quad 8 \quad 9$

We skip $\texttt{SA}[1]$ since $\mathcal{P}_0 = \emptyset$. Thus, $\mathcal{P}_6 = \{4, 5\}$ is inserted next.

$\boxed{12} \mid \boxed{0} \mid \boxed{6} \mid \boxed{10} \mid \boxed{4} \mid 1 \mid 7 \mid 3 \mid \boxed{11} \mid \boxed{5} \mid 2 \quad 8 \quad 9$

Next we have $\mathcal{P}_{10} = \{7, 9\}$.

$\boxed{12} \mid \boxed{0} \mid \boxed{6} \mid \boxed{10} \mid \boxed{4} \mid 1 \mid \boxed{7} \mid 3 \mid \boxed{11} \mid \boxed{5} \mid \boxed{9} \mid 2 \quad 8$

Next we have $\mathcal{P}_4 = \{1, 3\}$.

$\boxed{12} \mid \boxed{0} \mid \boxed{6} \mid \boxed{10} \mid \boxed{4} \mid \boxed{1} \mid \boxed{7} \mid \boxed{3} \mid \boxed{11} \mid \boxed{5} \mid \boxed{9} \mid 2 \quad 8$

The only remaining nonempty $\mathcal{P}_i$ are $\mathcal{P}_3 = \{2\}$ and $\mathcal{P}_9 = \{8\}$, which are considered in that order. Inserting them gives the suffix array.

$\boxed{12} \mid \boxed{0} \mid \boxed{6} \mid \boxed{10} \mid \boxed{4} \mid \boxed{1} \mid \boxed{7} \mid \boxed{3} \mid \boxed{11} \mid \boxed{5} \mid \boxed{9} \mid \boxed{2} \mid \boxed{8}$

Fig. 3. Refining a Lyndon grouping for $S = \texttt{acedcebceece\$}$ (see Fig. 2) into the suffix array, as done in Algorithm 1. Inserted elements are colored green. (Color figure online)

Note that each element $i \in [0..n-1)$ has exactly one next smaller suffix, hence there is exactly one j with $i \in \mathcal{P}_j$ and thus i is inserted exactly once into a new singleton group in Algorithm 1. Therefore, it suffices to map each group from the Lyndon grouping obtained from Phase I to its current start; we use an array C that contains the current group starts.

There are two major differences between our Phase II and Baier's, both are concerned with the iteration over the $\mathcal{P}_i$-sets.

The first difference is the way in which we determine the elements of $\mathcal{P}_i$ for some i. The following observations enable us to iterate over $\mathcal{P}_i$.

Lemma 2. *$\mathcal{P}_i$ is empty if and only if $i = 0$ or $S_{i-1} <_{lex} S_i$. Furthermore, if $\mathcal{P}_i \neq \emptyset$ then $i - 1 \in \mathcal{P}_i$.*

Lemma 3. *For some $j \in [0..i)$, we have $j \in \mathcal{P}_i$ if and only if j's last child is in $\mathcal{P}_i$, or $j = i - 1$ and $S_j >_{lex} S_i$.*

Specifically, (if $\mathcal{P}_i$ is not empty) we can iterate over $\mathcal{P}_i$ by walking up the $\texttt{pss}$-tree starting from $i-1$ and halting when we encounter a node that is not the last child of its parent.[2] Baier [1,2] tests whether $i - 1$ ($\texttt{pss}[j]$) is in $\mathcal{P}_i$ by explicitly checking whether $i - 1$ ($\texttt{pss}[j]$) has already been written to A using an explicit marker for each suffix. Reading and writing those markers leads to bad cache performance because the accessed memory locations are unpredictable (for the CPU/compiler). Lemmata 2 and 3 enable us to avoid reading and writing those markers. In fact, in our implementation of Phase II, the array A is the only memory written to that is not always in the cache. Lemma 2 tells us whether we need to follow the $\texttt{pss}$-chain starting at $i - 1$ or not. Namely, this is the case if and only if $S_{i-1} >_{lex} S_i$, i.e. $i - 1$ is a leaf in the $\texttt{pss}$-tree. This information is required when we encounter i in A during the outer for-loop in Algorithm 1, thus

[2] Note that $n-1$ is the last child of the artificial root -1. This ensures that we always halt before we actually reach the root of the $\texttt{pss}$-tree. Moreover, Corollary 1 implies that the order in which we process the elements in $\mathcal{P}_i$ is not important.

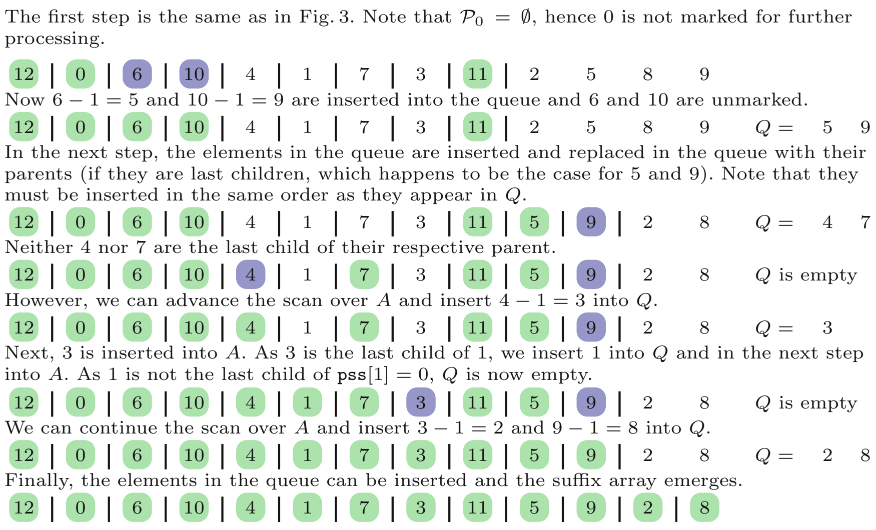
```
$A \leftarrow (n-1)\perp^{n-1}$ ; // set $A[0] = n-1$, fill the rest with "undefined"
$Q \leftarrow$ queue containing only $n-1$;
$i \leftarrow 1$; // current index in $A$
while $Q$ is not empty do
    $s \leftarrow Q.size()$;
    repeat $s$ times // insert elements that are currently in the queue
        $v \leftarrow Q.pop()$;
        if $pss[v]$ is marked then // $v$ is last child of pss$[v]$
            $Q.push(\text{pss}[v])$;
        end
        $A[C[G[v]]] \leftarrow v$; // insert $v$
        if $pss[v] + 1 < v$ then mark $A[C[G[v]]]$; // $v - 1$ is leaf
        $C[G[v]] \leftarrow C[G[v]] + 1$; // increment current start of $v$'s old group
    end
    while $Q.size() < w \wedge i < n \wedge A[i] \neq \perp$ do // refill the queue
        if $A[i]$ is marked then // $A[i] - 1$ is leaf
            $Q.push(A[i] - 1)$;
        end
        $i \leftarrow i + 1$;
    end
end
```

Algorithm 2: Breadth-first approach to Phase II. The constant w is the maximum queue size and $G[i]$ is the index of the group start pointer of i's group in C.

we *mark* such an entry i in A if and only if $\mathcal{P}_i \neq \emptyset$. Implementation-wise, we use the most significant bit (MSB) of an entry to indicate whether it is marked or not. By definition, we have $S_{i-1} >_{lex} S_i$ if and only if $\texttt{pss}[i] + 1 < i$. Since $\texttt{pss}[i]$ must be accessed anyway when i is inserted into A (for traversing the pss-chain), we can insert i marked or unmarked into A. Further, Lemma 3 implies that we must stop traversing a pss-chain when the current element is not the last child of its parent. We mark the entries in pss accordingly, also using the MSB of each entry. In the rest of this paper, we assume pss to be marked in this way.

Consider for instance $i = 6$ in our running example. As $6 - 1 = 5$ is a leaf (cf. Fig. 1), we have $5 \in \mathcal{P}_6$. We can deduce the fact that 5 is indeed a leaf from $\texttt{pss}[6] = 0 < 5$ alone. Further, 5 is the last child of $\texttt{pss}[5] = 4$, so $4 \in \mathcal{P}_6$. Since 4 is not the last child of $\texttt{pss}[4] = 0$, we have $\mathcal{P}_6 = \{4, 5\}$.

The second major change concerns the cache-unfriendliness of traversing the $\mathcal{P}_i$-sets. This bad cache performance results from the fact that the next pss-value (and the group start pointer) cannot be fetched until the current one is in memory. Instead of traversing the $\mathcal{P}_i$-sets one after another, we opt to traversing multiple such sets in a sort of breadth-first-search manner simultaneously. Specifically, we maintain a small ($\leq 2^{10}$ elements) queue Q of elements (nodes in the pss-tree) that can currently be processed. Then we iterate over Q and process the entries one after another. Parents of last children are inserted into Q in the same order as the respective children. After each iteration, we continue to scan over A and for each encountered marked entry i insert $i-1$ into Q until we either encounter an empty entry in A or Q reaches its maximum capacity. This is repeated until the suffix array emerges. The queue size could be unlimited, but limiting it ensures that it fits into the CPU's cache. Figure 4 shows our Phase II on the running example and Algorithm 2 describes it formally in pseudo code.



The first step is the same as in Fig. 3. Note that $\mathcal{P}_0 = \emptyset$, hence 0 is not marked for further processing.

12 | 0 | 6 | 10 | 4 | 1 | 7 | 3 | 11 | 2 5 8 9

Now $6-1=5$ and $10-1=9$ are inserted into the queue and 6 and 10 are unmarked.

12 | 0 | 6 | 10 | 4 | 1 | 7 | 3 | 11 | 2 5 8 9 $Q =$ 5 9

In the next step, the elements in the queue are inserted and replaced in the queue with their parents (if they are last children, which happens to be the case for 5 and 9). Note that they must be inserted in the same order as they appear in Q.

12 | 0 | 6 | 10 | 4 | 1 | 7 | 3 | 11 | 5 | 9 | 2 8 $Q =$ 4 7

Neither 4 nor 7 are the last child of their respective parent.

12 | 0 | 6 | 10 | 4 | 1 | 7 | 3 | 11 | 5 | 9 | 2 8 Q is empty

However, we can advance the scan over A and insert $4-1=3$ into Q.

12 | 0 | 6 | 10 | 4 | 1 | 7 | 3 | 11 | 5 | 9 | 2 8 $Q =$ 3

Next, 3 is inserted into A. As 3 is the last child of 1, we insert 1 into Q and in the next step into A. As 1 is not the last child of $\texttt{pss}[1] = 0$, Q is now empty.

12 | 0 | 6 | 10 | 4 | 1 | 7 | 3 | 11 | 5 | 9 | 2 8 Q is empty

We can continue the scan over A and insert $3-1=2$ and $9-1=8$ into Q.

12 | 0 | 6 | 10 | 4 | 1 | 7 | 3 | 11 | 5 | 9 | 2 8 $Q =$ 2 8

Finally, the elements in the queue can be inserted and the suffix array emerges.

12 | 0 | 6 | 10 | 4 | 1 | 7 | 3 | 11 | 5 | 9 | 2 | 8

Fig. 4. Refining a Lyndon grouping for $S = acedcebceece\$$ (see Fig. 2) into the suffix array using Algorithm 2. Marked entries are coloured blue while inserted but unmarked elements are coloured green. Note that the uncoloured entries are not actually present in the array A but only serve to indicate the current Lyndon grouping. (Color figure online)

Theorem 1. *Algorithm 2 correctly computes the suffix array from a Lyndon grouping.*

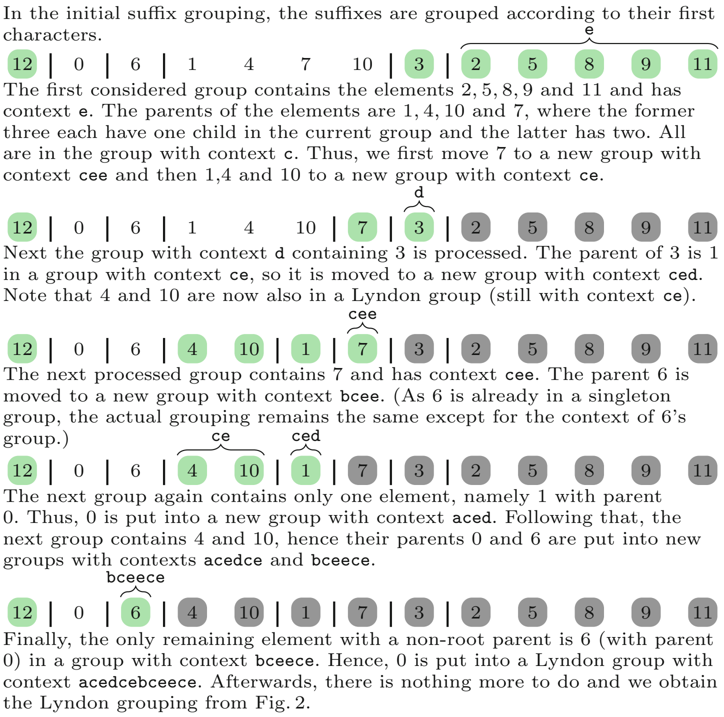
3.2 Phase I

In Phase I, a Lyndon grouping is derived from a suffix grouping in which the group contexts have length (at least) one. That is, the suffixes are sorted and grouped by their Lyndon prefixes. Lemma 4 describes the relationship between the Lyndon prefixes and the pss-tree that is essential to Phase I.

Lemma 4. *Let $c_1 < \cdots < c_k$ be the children of $i \in [0..n)$ in the* $\texttt{pss}$*-tree. $\mathcal{L}_i$ is $S[i]$ concatenated with the Lyndon prefixes of $c_1, \ldots, c_k$. More formally:*

$$\mathcal{L}_i = S\left[i..\texttt{nss}[i]\right) = S[i]S\left[c_1..c_2\right)\ldots S\left[c_k..\texttt{nss}[i]\right) = S[i]\mathcal{L}_{c_1}\ldots\mathcal{L}_{c_k}$$

We start from the *initial suffix grouping* in which the suffixes are grouped according to their first characters. From the relationship between the Lyndon prefixes and the pss-tree in Lemma 4 one can get the general idea of extending the context of a node's group with the Lyndon prefixes of its children (in correct order) while maintaining the sorting [1]. Note that any node is by definition in a higher group than its parent. Also, by Lemma 4 the leaves of the pss-tree are already in Lyndon groups in the initial suffix grouping. Therefore, if we consider the groups in lexicographically decreasing order (i.e. higher to lower) and append the context of the current group to each parent (and insert



In the initial suffix grouping, the suffixes are grouped according to their first characters.

The first considered group contains the elements 2, 5, 8, 9 and 11 and has context `e`. The parents of the elements are 1, 4, 10 and 7, where the former three each have one child in the current group and the latter has two. All are in the group with context `c`. Thus, we first move 7 to a new group with context `cee` and then 1,4 and 10 to a new group with context `ce`.

Next the group with context `d` containing 3 is processed. The parent of 3 is 1 in a group with context `ce`, so it is moved to a new group with context `ced`. Note that 4 and 10 are now also in a Lyndon group (still with context `ce`).

The next processed group contains 7 and has context `cee`. The parent 6 is moved to a new group with context `bcee`. (As 6 is already in a singleton group, the actual grouping remains the same except for the context of 6's group.)

The next group again contains only one element, namely 1 with parent 0. Thus, 0 is put into a new group with context `aced`. Following that, the next group contains 4 and 10, hence their parents 0 and 6 are put into new groups with contexts `acedce` and `bceece`.

Finally, the only remaining element with a non-root parent is 6 (with parent 0) in a group with context `bceece`. Hence, 0 is put into a Lyndon group with context `acedcebceece`. Afterwards, there is nothing more to do and we obtain the Lyndon grouping from Fig. 2.

Fig. 5. Refining the initial suffix grouping for $S = abccabccbcc\$$ (see Fig. 2) into the Lyndon grouping. Elements in Lyndon groups are marked gray or green, depending on whether they have been processed already. Note that the applied procedure does not entirely correspond to our algorithm for Phase I; it only serves to illustrate the sorting principle. (Color figure online)

them into new groups accordingly), each encountered group is guaranteed to be Lyndon [1]. Consequently, we obtain a Lyndon grouping. Figure 5 shows this principle applied to our running example. Formally, the suffix grouping satisfies the following property during Phase I before and after processing a group:

Property 1. For any $i \in [0..n)$ with children $c_1 < \cdots < c_k$ there is $j \in [0..k]$ such that (a) $c_1, \ldots, c_j$ are in groups that have already been processed, (b) $c_{j+1}, \ldots, c_k$ are in groups that have not yet been processed, and (c) the context of the group containing i is $S[i]\mathcal{L}_{c_1} \ldots \mathcal{L}_{c_j}$. Furthermore, each processed group is Lyndon.

Additionally and unlike in Baier's original approach, all groups created during our Phase I are either Lyndon or only contain elements whose Lyndon prefix is different from the group's context.

Definition 4 (Strongly preliminary group). *We call a preliminary group* $\mathcal{G} = \langle g_s, g_e, |\alpha| \rangle$ strongly preliminary *if and only if* $\mathcal{G}$ *contains only elements* *whose Lyndon prefix is not α. A preliminary group that is not strongly preliminary is called* weakly preliminary*.*

Lemma 5. *For any weakly preliminary group $\mathcal{G} = \langle g_s, g_e, |\alpha| \rangle$ there is some $g' \in [g_s \mathinner{..} g_e)$ such that $\mathcal{G}' = \langle g_s, g', |\alpha| \rangle$ is a Lyndon group and $\mathcal{G}'' = \langle g'+1, g_e, |\alpha| \rangle$ is a strongly preliminary group.*

For instance, in Fig. 5 there is a group containing 1,4 and 10 with context `ce`. However, 4 and 10 have this context as Lyndon prefix while 1 has `ced`. Consequently, 1 will later be moved to a new group. Hence, when Baier (and Bertram et al.) create a weakly preliminary group (in Fig. 5 this happens while processing the Lyndon group with context `e`), we instead create two groups, the lower containing 4 and 10 and the higher containing 1.

During Phase I we maintain the suffix grouping using the following data structures. Two arrays A and I of length n each, where A contains the unprocessed Lyndon groups and the sizes of the strongly preliminary groups, and I maps each element $s \in [0 \mathinner{..} n)$ to the start of the group containing s. We call $I[s]$ the *group pointer* of s. Further, we store the starts of the already processed groups in a list C . Let $\mathcal{G} = \langle g_s, g_e, |\alpha| \rangle$ be a group. For each $s \in \mathcal{G}$ we have $I[s] = g_s$. If $\mathcal{G}$ is Lyndon and has not yet been processed, we also have $s \in A\,[g_s \mathinner{..} g_e]$ for all $s \in \mathcal{G}$ and $A[g_s] < A[g_s+1] < \cdots < A[g_e]$. If $\mathcal{G}$ is Lyndon and has been processed already, there is some j such that $C[j] = g_s$. If $\mathcal{G}$ is (strongly) preliminary we have $A[g_s] = g_e + 1 - g_s$ and $A[k] = 0$ for all $k \in (g_s \mathinner{..} g_e]$.

There are several reasons why our Phase I is much faster than Baier's. Firstly, we do not write preliminary groups to A. Secondly, we compute `pss` beforehand using an algorithm by Bille et al. [4] instead of on the fly as Baier does [1,2]. Furthermore, we have the Lyndon groups in A sorted and store the sizes of the strictly preliminary groups in A as well. The former makes finding the number of children a parent has in the currently processed group easier and faster. The latter makes the separate array of length n used by Baier [1,2] for the group sizes obsolete and is made possible by the fact that we only write Lyndon groups to A. For reasons why these changes lead to a faster algorithm see [13].

As alluded above, we follow Baier's general approach and consider the Lyndon groups in lexicographically decreasing order while updating the groups containing the parents of elements in the current group. Since children are in higher groups than their parents by definition, when we encounter some group $\mathcal{G} = \langle g_s, g_e, |\alpha| \rangle$, the children of any element in $\mathcal{G}$ are in already processed groups. Hence, by Property 1 $\mathcal{G}$ must be Lyndon. For a formal proof see [1].

In the rest of this section we explain how to actually process a Lyndon group.

Let $\mathcal{G} = \langle g_s, g_e, |\alpha| \rangle$ be the currently processed group and w.l.o.g. assume that no element in $\mathcal{G}$ has the root -1 as parent (we do not have the root in the suffix grouping, thus nodes with the root as parent can be ignored here). Furthermore, let $\mathcal{A}$ be the set of parents of elements in $\mathcal{G}$ (i.e. $\mathcal{A} = \{\texttt{pss}[i] : i \in \mathcal{G}, \texttt{pss}[i] \geq 0\}$) and let $\mathcal{G}_1 < \cdots < \mathcal{G}_k$ be those (necessarily preliminary) groups containing elements from $\mathcal{A}$. For each $g \in [1 \mathinner{..} k]$ let α_g be the context of $\mathcal{G}_g$.

As noted in Fig. 5, we have to consider the number of children an element in $\mathcal{A}$ has in $\mathcal{G}$. Specifically, we need to move two parents in $\mathcal{A}$ which are currently in the same group to different new groups if they have differing numbers of children in $\mathcal{G}$. Let $\mathcal{A}_\ell$ contain those elements from $\mathcal{A}$ with exactly ℓ children in $\mathcal{G}$. Maintaining Property 1 requires that, after processing $\mathcal{G}$, for some $g \in [1..k]$ the elements in $\mathcal{G}_g \cap \mathcal{A}_\ell$ are in groups with context $\alpha_g\alpha^\ell$. For any $\ell < \ell'$, we have $\alpha_g\alpha^\ell <_{lex} \alpha_g\alpha^{\ell'}$, thus the elements in $\mathcal{G}_g \cap \mathcal{A}_\ell$ must form a lower group than those in $\mathcal{G}_g \cap \mathcal{A}_{\ell'}$ after $\mathcal{G}$ has been processed [1,2]. To achieve this, first the parents in $\mathcal{A}_{|\mathcal{G}|}$ are moved to new groups immediately following their respective old groups, then those in $\mathcal{A}_{|\mathcal{G}|-1}$ and so on [1,2].

We proceed as follows. First, determine $\mathcal{A}$ and count how many children each parent has in $\mathcal{G}$. Then, sort the parents according to these counts using a bucket sort.[3] Further, partition the elements in each bucket into two sub-buckets depending on whether they should be inserted into Lyndon groups or strongly preliminary groups. Then, for the sub-buckets (in the order of decreasing count; for equal counts: first strongly preliminary then Lyndon sub-buckets) move the parents into new groups.[4] Because of space constraints, we do not describe the rather technical details. These can be found in the extended paper [13].

4 Experiments

Our implementation `FGSACA` of the optimised `GSACA` is publicly available.[5]

We compare our algorithm with the `GSACA` implementation by Baier [1,2], and the `double sort` algorithms `DS1` and `DSH` by Bertram et al. [3]. The latter two also use the grouping principle but employ integer sorting and have super-linear time complexity. `DSH` differs from `DS1` only in the initialisation: in `DS1` the suffixes are sorted by their first character while in `DSH` up to 8 characters are considered. We further include `DivSufSort` 2.0.2 and `libsais` 2.7.1 since the former is used by Bertram et al. as a reference [3] and the latter is the currently fastest suffix sorter known to us.

The algorithms were evaluated on real texts (in the following `PC-Real`), real repetitive texts (`PC-Rep-Real`) and artificial repetitive texts (`PC-Rep-Art`) from the Pizza & Chili corpus. To test the algorithms on texts for which a 32-bit suffix array is not sufficient, we also included larger texts (`Large`), namely the first 10^{10} bytes from the English Wikipedia dump from 01.06.2022 and the human DNA concatenated with itself. For more detail on the data and our testing methodology see the longer version of this paper [13].

All algorithms were faster on the more repetitive datasets, on which the differences between the algorithms were also smaller. On all datasets, our algorithm

[3] Note that the sum of the counts is $|\mathcal{G}|$, hence the time complexity of the bucket sort is linear in the size of the group.

[4] Note that Baier broadly follows the same steps (determine parents, sort them, move them to new groups accordingly) [1,2]. However, each individual step is different because of our distinction between strongly preliminary, weakly preliminary and Lyndon groups.

[5] https://gitlab.com/qwerzuiop/lfgsaca.

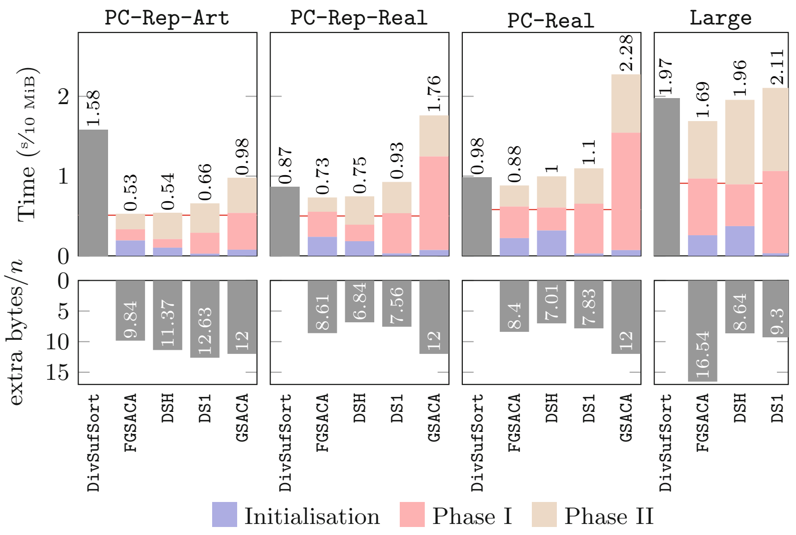


Fig. 6. Normalised running time and working memory averaged for each category. The horizontal red line indicates the time for `libsais`. For `Large` we did not test `GSACA` because Baier's reference implementation only supports 32-bit words. (Color figure online)

is between 46% and 60% faster than `GSACA` and compared to `DSH` about 2% faster on repetitive data, over 11% faster on `PC-Real` and over 13% faster on `Large`.

Especially notable is the difference in the time required for Phase II: Our Phase II is between 33% and 50% faster than Phase II of `DSH`. Our Phase I is also faster than Phase I of `DS1` by a similar margin. Conversely, Phase I of `DSH` is much faster than our Phase I. However, this is only due to the more elaborate construction of the initial suffix grouping as demonstrated by the much slower Phase I of `DS1`. Compared to `FGSACA`, `libsais` is between 46% and 3% faster.

Memory-wise, for 32-bit words, `FGSACA` uses about 8.83 bytes per input character, while `DS1` and `DSH` use 8.94 and 8.05 bytes/character, respectively. `GSACA` always uses 12 bytes/character. On `Large`, `FGSACA` expectedly requires about twice as much memory. For `DS1` and `DSH` this is not the case, mostly because they use 40-bit integers for the additional array of length n that they require (while we use 64-bit integers). `DivSufSort` requires only a small constant amount of working memory and `libsais` never exceeded 21kiB of working memory on our test data.

Acknowledgements. This work was supported by the Deutsche Forschungsgemeinschaft (DFG - German Research Foundation) (OH 53/7-1).

A Proofs

Lemma 1. *For any $j, j' \in \mathcal{P}_i$ we have $\mathcal{L}_j \neq \mathcal{L}_{j'}$ if and only if $j \neq j'$.*

Proof. Let $j, j' \in \mathcal{P}_i$ and $j \neq j'$. By definition of $\mathcal{P}_i$ we have $\texttt{nss}[j] = \texttt{nss}[j'] = i$. Since $\mathcal{L}_j = S\,[j\,..\,\texttt{nss}[j])$ and $\mathcal{L}_{j'} = S\,[j'\,..\,\texttt{nss}[j'])$, $\mathcal{L}_j$ and $\mathcal{L}_{j'}$ have different lengths, implying the claim.

Lemma 2. *$\mathcal{P}_i$ is empty if and only if $i = 0$ or $S_{i-1} <_{lex} S_i$. Furthermore, if $\mathcal{P}_i \neq \emptyset$ then $i - 1 \in \mathcal{P}_i$.*

Proof. $\mathcal{P}_0 = \emptyset$ by definition. Let $i \in [1\,..\,n)$. If $S_{i-1} >_{lex} S_i$ we have $\texttt{nss}[i-1] = i$ and thus $i - 1 \in \mathcal{P}_i$. Otherwise ($S_{i-1} <_{lex} S_i$), assume there is some $j < i - 1$ such that $\texttt{nss}[j] = i$. By definition, $S_j >_{lex} S_i$ and $S_j <_{lex} S_k$ for each $k \in (j\,..\,i)$. But by transitivity we also have $S_j >_{lex} S_{i-1}$, which is a contradiction, hence $\mathcal{P}_i$ must be empty.

Lemma 3. *For some $j \in [0\,..\,i)$, we have $j \in \mathcal{P}_i$ if and only if j's last child is in $\mathcal{P}_i$, or $j = i - 1$ and $S_j >_{lex} S_i$.*

Proof. By Lemma 2 we may assume $\mathcal{P}_i \neq \emptyset$ and $j + 1 < i$, otherwise the claim is trivially true. If j is a leaf we have $\texttt{nss}[j] = j + 1 < i$ and thus $j \notin \mathcal{P}_i$ by definition. Hence assume j is not a leaf and has $j' > j$ as last child, i.e. $\texttt{pss}[j'] = j$ and there is no $k > j'$ with $\texttt{pss}[k] = j$. It suffices to show that $j' \in \mathcal{P}_i$ if and only if $j \in \mathcal{P}_i$. Note that $\texttt{pss}[j'] = j$ implies $\texttt{nss}[j] > j'$.

$\Longrightarrow$: From $\texttt{nss}[j'] = i$ and thus $S_k >_{lex} S_{j'} >_{lex} S_j$ (for all $k \in (j'\,..\,i)$) we have $\texttt{nss}[j] \geq i$. Assume $\texttt{nss}[j] > i$. Then $S_i >_{lex} S_j$ and thus $\texttt{pss}[i] = j$, which is a contradiction.

$\Longleftarrow$: From $S_i <_{lex} S_j <_{lex} S_{j'}$ we have $\texttt{nss}[j'] \leq i$. Assume $\texttt{nss}[j'] < i$ for a contradiction. For all $k \in (j\,..\,j')$, $\texttt{pss}[j'] = j$ implies $S_k >_{lex} S_{j'}$. Furthermore, for all $k \in [j'\,..\,\texttt{nss}[j'])$ we have $S_k >_{lex} S_{\texttt{nss}[j']}$ by definition. In combination this implies $S_k >_{lex} S_{\texttt{nss}[j']}$ for all $k \in (j\,..\,\texttt{nss}[j'])$. As $\texttt{nss}[j] = i > \texttt{nss}[j']$ we hence have $\texttt{pss}[\texttt{nss}[j']] = j$, which is a contradiction.

Theorem 1. *Algorithm 2 correctly computes the suffix array from a Lyndon grouping.*

Proof. By Lemmata 2 and 3, Algorithms 1 and 2 are equivalent for a maximum queue size of 1. Therefore it suffices to show that the result of Algorithm 2 is independent of the queue size. Assume for a contradiction that the algorithm inserts two elements i and j with $S_i <_{lex} S_j$ belonging to the same Lyndon group with context α, but in a different order as Algorithm 1 would. This can only happen if j is inserted earlier than i. Note that, since i and j have the same Lyndon prefix α, the $\texttt{pss}$-subtrees T_i and T_j rooted at i and j, respectively, are isomorphic (see [4]). In particular, the path from the rightmost leaf in T_i to i has the same length as the path from the rightmost leaf in T_j to j. Thus, i and j are inserted in the same order as $S_{i+|\alpha|}$ and $S_{j+|\alpha|}$ occur in the suffix array. Now the claim follows inductively.

Lemma 4. *Let $c_1 < \cdots < c_k$ be the children of $i \in [0\mathinner{..}n)$ in the* pss*-tree. $\mathcal{L}_i$ is $S[i]$ concatenated with the Lyndon prefixes of $c_1, \ldots, c_k$. More formally:*

$$\mathcal{L}_i = S\left[i\mathinner{..}\texttt{nss}[i]\right) = S[i]S\left[c_1\mathinner{..}c_2\right)\ldots S\left[c_k\mathinner{..}\texttt{nss}[i]\right) = S[i]\mathcal{L}_{c_1}\ldots\mathcal{L}_{c_k}$$

Proof. By definition we have $\mathcal{L}_i = S\left[i\mathinner{..}\texttt{nss}[i]\right)$. Assume i has $k \geq 1$ children $c_1 < \cdots < c_k$ in the pss-tree (otherwise $\texttt{nss}[i] = i + 1$ and the claim is trivial). For the last child c_k we have $\texttt{nss}[c_k] = \texttt{nss}[i]$ from Lemma 3. Let $j \in [1\mathinner{..}k)$ and assume $\texttt{nss}[c_j] \neq c_{j+1}$. Then we have $\texttt{nss}[c_j] < c_{j+1}$, otherwise c_{j+1} would be a child of c_j. As we have $S_{\texttt{nss}[c_j]} <_{lex} S_{c_j}$ and $S_{c_j} <_{lex} S_{c_{j'}}$ for each $j' \in [1\mathinner{..}j)$ (by induction), we also have $S_{\texttt{nss}[c_j]} <_{lex} S_{i'}$ for each $i' \in (i\mathinner{..}\texttt{nss}[c_j])$. Since $\texttt{nss}[i] > \texttt{nss}[c_j]$, $\texttt{nss}[c_j]$ must be a child of i in the pss-tree, which is a contradiction.

Lemma 5. *For any weakly preliminary group $\mathcal{G} = \langle g_s, g_e, |\alpha|\rangle$ there is some $g' \in [g_s\mathinner{..}g_e)$ such that $\mathcal{G}' = \langle g_s, g', |\alpha|\rangle$ is a Lyndon group and $\mathcal{G}'' = \langle g'+1, g_e, |\alpha|\rangle$ is a strongly preliminary group.*

Proof. Let $\mathcal{G} = \langle g_s, g_e, |\alpha|\rangle$ be a weakly preliminary group. Let $F \subset \mathcal{G}$ be the set of elements from $\mathcal{G}$ whose Lyndon prefix is α. By Lemma 6 (below) we have $S_i <_{lex} S_j$ for any $i \in F, j \in \mathcal{G} \setminus F$. Hence, splitting $\mathcal{G}$ into two groups $\mathcal{G}' = \langle g_s, g_s + |F| - 1, |\alpha|\rangle$ and $\mathcal{G}'' = \langle g_s + |F|, g_e, |\alpha|\rangle$ results in a valid suffix grouping. Note that, by construction, the former is a Lyndon group and the latter is strongly preliminary.

Lemma 6. *For strings wu and wv over Σ with $u <_{lex} wu$ and $v >_{lex} wv$ we have $wu <_{lex} wv$.*

Proof. Note that there is no j such that $wv = w^j$ since otherwise v would be a prefix of wv and $v <_{lex} wv$ would hold. Hence, there are $k \in \mathbb{N}, \ell \in [0\mathinner{..}|w|), b \in \Sigma$ and $m \in \Sigma^*$ such that $wv = w^k w\left[0\mathinner{..}\ell\right) bm$ and $b > w[\ell]$. There are two cases:

– $wu = w^j$ for some $j \geq 1$
  - If $j\,|w| \leq k\,|w| + \ell$, then wu is a prefix of wv.
  - Otherwise, the first different symbol in wu and wv is at index $p = k\,|w| + \ell$ and we have $(wu)[p] = w^j[p] = w[\ell] < b = (wv)[p]$.

– There are $i \in \mathbb{N}, j \in [0\mathinner{..}|w|), a \in \Sigma$ and $q \in \Sigma^*$ such that $wu = w^i w\left[0\mathinner{..}j\right) aq$ and $a < w[j]$.
  - If $|w^i w\left[0\mathinner{..}j\right)| \leq |w^k w\left[0\mathinner{..}\ell\right)|$, the first different symbol is at position $p = |w^i w\left[0\mathinner{..}j\right)|$ with $wu[p] = a < w[j] \leq wv[p]$.
  - Otherwise, the first different symbol is at position $p = |w^k w\left[0\mathinner{..}\ell\right)|$ with $wv[p] = b > w[\ell] = wu[p]$.

In all cases, the claim follows.

References

1. Baier, U.: Linear-time suffix sorting. Master's thesis, Ulm University (2015)
2. Baier, U.: Linear-time suffix sorting - a new approach for suffix array construction. In: Grossi, R., Lewenstein, M. (eds.) 27th Annual Symposium on Combinatorial Pattern Matching. Leibniz International Proceedings in Informatics, vol. 54. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016)
3. Bertram, N., Ellert, J., Fischer, J.: Lyndon words accelerate suffix sorting. In: Mutzel, P., Pagh, R., Herman, G. (eds.) 29th Annual European Symposium on Algorithms. Leibniz International Proceedings in Informatics, vol. 204, pp. 15:1–15:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
4. Bille, P., et al.: space efficient construction of Lyndon arrays in linear time. In: Czumaj, A., Dawar, A., Merelli, E. (eds.) 47th International Colloquium on Automata, Languages, and Programming. Leibniz International Proceedings in Informatics, vol. 168, pp. 14:1–14:18. Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2020)
5. Fischer, J., Kurpicz, F.: Dismantling DivSufSort. In: Holub, J., Žd'árek, J. (eds.) Proceedings of the Prague Stringology Conference 2017, pp. 62–76 (2017)
6. Franek, F., Paracha, A., Smyth, W.F.: The linear equivalence of the suffix array and the partially sorted Lyndon array. In: Holub, J., Žd'árek, J. (eds.) Proceedings of the Prague Stringology Conference 2017, pp. 77–84 (2017)
7. Goto, K.: Optimal time and space construction of suffix arrays and LCP arrays for integer alphabets. In: Holub, J., Žd'árek, J. (eds.) Proceedings of the 23rd Prague Stringology Conference, pp. 111–125 (2017)
8. Li, Z., Li, J., Huo, H.: Optimal in-place suffix sorting. Inf. Comput. **285**, 104818 (2022). https://doi.org/10.1016/j.ic.2021.104818. ISSN 0890-5401
9. 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. Society for Industrial and Applied Mathematics (1990)
10. Nong, G.: Practical linear-time O(1)-workspace suffix sorting for constant alphabets. ACM Trans. Inf. Syst. **31**(3), 1–15 (2013)
11. Nong, G., Zhang, S., Chan, W.H.: Linear suffix array construction by almost pure induced-sorting. In: 2009 Data Compression Conference, pp. 193–202 (2009)
12. Ohlebusch, E.: Bioinformatics algorithms: sequence analysis, genome rearrangements, and phylogenetic reconstruction. Oldenbusch Verlag (2013)
13. Olbrich, J., Ohlebusch, E., Büchler, T.: On the optimisation of the GSACA suffix array construction algorithm (2022). https://doi.org/10.48550/ARXIV.2206.12222
14. Pierre Duval, J.: Factorizing words over an ordered alphabet. J. Algorithms **4**(4), 363–381 (1983)