Linear-time Suffix Sorting – A New Approach for Suffix Array Construction

Uwe Baier

Institute of Theoretical Computer Science, Ulm University
D-89069 Ulm, Germany
uwe.baier@uni-ulm.de

Abstract

This paper presents a new approach for linear-time suffix sorting. It introduces a new sorting principle that can be used to build the first non-recursive linear-time suffix array construction algorithm named GSACA. Although GSACA cannot keep up with the performance of state of the art suffix array construction algorithms, the algorithm introduces a couple of new ideas for suffix array construction, and therefore can be seen as an 'idea collection' for further suffix array construction improvements.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Suffix array, sorting algorithm, linear-time

Digital Object Identifier 10.4230/LIPIcs.CPM.2016.23

1 Introduction

The suffix array is an elementary data structure used in string processing as well as in data compression. Introduced by Manber and Myers in 1990 [11], the suffix array nowadays finds application in dozens of different areas. Constructing a suffix array from a given string unfortunately turns out to be a computationally hard task; despite the existence of linear-time algorithms for suffix array construction, some super-linear algorithms still achieve better results in practice.

As data grows bigger and bigger, 'optimal' suffix array construction algorithms (SACAs) nowadays still stay an area of great interest. According to a survey paper of Puglisi et al. [19], an 'optimal' SACA fulfils three requirements: First, an algorithm should run in asymptotic minimal worst-case-time, linear-time in an optimal way. Second, an algorithm should run fast in practice, too. Finally, the algorithm should consume as less extra space in addition to the text and the suffix array as possible, a constant amount optimally.

Presently, no SACA is able to meet all of those requirements in an optimal way. Our contribution towards this goal will be the presentation of a new design principle for suffix array construction, resulting in the first non-recursive linear-time suffix array construction algorithm. Although the new algorithm is not able to fulfil all requirements of optimal suffix array construction, it presents a new approach for suffix array construction, and therefore is interesting from a theoretical point of view.

Overview This paper will be organised as follows: Section 2 contains a short introduction to suffix arrays and basic definitions. Section 3 presents the new sorting principle along with an introductory example, before Section 4 lists the new algorithm with explanations of technical details. Section 5 contains performance analyses of the new algorithm, before Section 6 summarises the results and gives an outline for future work.

27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016).
Editors: Roberto Grossi and Moshe Lewenstein; Article No. 23; pp. 23:1–23:12
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Related Work The suffix array first was described in 1990 by Manber and Myers [11] as a space-saving alternative to suffix trees [21].

Then, in 2003, four linear-time[1] SACAs were contemporary introduced by Kim et al. [8], Kärkkäinen and Sanders [7], Ko and Aluru [10] and Hon et al. [6], before Joong Chae Na introduced another linear-time SACA in 2005 [15]. Two algorithms stood out: the *Skew Algorithm* by Kärkkäinen and Sanders [7] because of its elegance, as well as the algorithm by Ko and Aluru [10] because of its good performance in practice.

Later on, in 2009, Nong et al. presented two new algorithms using the induced sorting principle [17, 18] as an improvement to the algorithm by Ko and Aluru. One of those algorithms, called *SA-IS* [17], was able to outperform most of other existing SACAs [14] while guaranteeing asymptotic linear runtime and almost optimal space requirements. In the meantime, performance of *SA-IS* was further improved while decreasing the required workspace to an only alphabet-dependent linear term [16]. Consequently, variants of the *SA-IS* algorithm serve as best linear-time SACAs known at the moment.

2 Preliminaries

Let Σ be a totally ordered set (alphabet) of elements (characters). A string S of length n over alphabet Σ is a finite sequence of n characters originating from Σ. The empty string with length 0 is denoted by ε.

Let i and j be two integers in range $[1, n]$. We denote by

- $S[i]$ the i-th character of S.
- $S[i..j]$ the substring of S starting at the i-th and ending at the j-th position. We state $S[i..j] = \varepsilon$ if $i > j$, and define $S[i..j+1) = S[i..j]$.
- S_i the suffix of S starting at the i-th position, i.e. $S_i = S[i..n]$.

Furthermore, we call S a *nullterminated* string if $\$ \in \Sigma$, $\$ < c$ for all $c \in \Sigma \setminus \{\$\}$, and $\$$ occurs exactly once in S, at the end of the string. First, a definition of the suffix array shall be presented. Additionally, next lexicographically smaller suffixes are required.

▶ **Definition 1.** Let Σ be an alphabet, S be a string of length n over alphabet Σ and T be a string of length m over alphabet Σ. We write $S <_{\mathsf{lex}} T$ and say that S is *lexicographically smaller* than T, if one of the following conditions holds:

- There exists an i $(1 \le i \le \min\{n, m\})$ with $S[i] < T[i]$ and $S[1..i) = T[1..i)$.
- S is a *proper prefix* of T, i.e. $n < m$ and $S[1..n] = T[1..n]$.

▶ **Definition 2.** Let S be a nullterminated string of length n. The *suffix array* SA of S is a permutation of integers in range $[1, n]$ satisfying $S_{\mathsf{SA}[1]} <_{\mathsf{lex}} S_{\mathsf{SA}[2]} <_{\mathsf{lex}} \cdots <_{\mathsf{lex}} S_{\mathsf{SA}[n]}$. The *inverse suffix array* ISA is the inverse permutation of SA.

▶ **Definition 3.** Let S be a nullterminated string of length n, and let i be an integer in range $[1, n)$. Then, by $\widehat{i}$ we denote the position of the next lexicographically smaller suffix of S_i, i.e. $\widehat{i} \coloneqq \min\{\ j \in [i \dots n] \mid S_j <_{\mathsf{lex}} S_i\ \}$. Also, we define $\widehat{n} \coloneqq n + 1$ for the last suffix of S.[2]

An example of these definitions can be found in Table 1.

[1] Super-linear-time SACAs are not object of interest here; we refer to the survey paper of Puglisi et al. [19] for more information about them.

[2] One can think of this as follows: if we define an imaginary empty last suffix $S_{n+1} \coloneqq \varepsilon$, then S_{n+1} is a proper prefix of S_n, so S_{n+1} is the next smaller suffix of S_n.

Table 1 Suffix array and next lexicographically smaller suffixes of $S = \texttt{graindraining\$}$.

i	$\mathsf{SA}[i]$	$\widehat{\mathsf{SA}[i]}$	$S_{\mathsf{SA}[i]}$	$S[\mathsf{SA}[i]..\widehat{\mathsf{SA}[i]})$
1	14	15	$	$
2	3	14	aindraining$	aindraining
3	8	14	aining$	aining
4	6	8	draining$	dr
5	13	14	g$	g
6	1	3	graindraining$	gr
7	4	6	indraining$	in
8	11	13	ing$	in
9	9	11	ining$	in
10	5	6	ndraining$	n
11	12	13	ng$	n
12	10	11	ning$	n
13	2	3	raindraining$	r
14	7	8	raining$	r

3 Algorithmic Idea

Within this Section, the algorithmic idea of the new algorithm will be presented. The main idea is to split the suffix array construction in two phases.

In a first phase, suffixes are divided into suffix groups as if each suffix S_i consists only of the string $S[i..\widehat{i})$: If $S[i..\widehat{i}) = S[j..\widehat{j})$ holds for two suffixes S_i and S_j, then they belong to the same group, otherwise to different groups. For any group $\mathcal{G}$ containing a suffix S_i, we denote the string $S[i..\widehat{i})$ as the *group context* of $\mathcal{G}$. In addition to the division of suffixes, the groups itself also will be ordered by comparing their group contexts. When comparing suffix groups by their contexts, the terms 'lower group' and 'higher group' will be used rather than the terms 'smaller' or 'larger', because groups are sets, and the latter both terms usually refer to set sizes, not to lexicographic comparison.

Afterwards, in a second phase, this group structure can be used to compute the suffix array. By iterating over the suffix array in ascending lexicographic order and completing the contexts of suffixes such that only groups with a single suffix remain, the desired order of suffixes can be obtained. A sketch of the principle can be found in Algorithm 1.

First, let's clarify the correctness of the principle by some argumentation. Assume that before the i-th iteration of the outer loop in Phase 2 (lines 4 to 8) all entries $\mathsf{SA}[1] \cdots \mathsf{SA}[i]$ were computed correctly. Then, within the i-th iteration, each further computed SA-entry is correct: Let j be any index with $\widehat{j} = \mathsf{SA}[i]$. Assume that an index k from the same group exists such that $S_k <_{\mathsf{lex}} S_j$. Because $\text{group}(k) = \text{group}(j)$, by the sorting in Phase 1, $S[j..\widehat{j}) = S[k..\widehat{k})$ holds, so $S_{\widehat{k}} <_{\mathsf{lex}} S_{\widehat{j}}$ must hold. Because of the ascending iteration order of the outer loop in Phase 2, $\widehat{k}$ must have been processed in one of the previous $i-1$ iterations. Within this iteration, the index k was processed in the inner loop of Phase 2, and thus has been removed from its group in line 8, $\text{group}(k) \neq \text{group}(j)$, contradiction. For the same reason, and because of the group order computed in Phase 1 (line 2), exactly those suffixes S_k with $\text{group}(k) < \text{group}(j)$ must be lexicographically smaller than S_j, so j is correctly placed into the suffix array in line 7.

Now we know that all entries are placed correctly to SA, but it remains to show that the suffix array is filled entirely. Therefore, consider the point in time after the i-th iteration of the outer loop in Phase 2, and let S_j be the lexicographically $i+1$-th smallest suffix.

Algorithm 1 Suffix array construction for a given nullterminated string S of length n.

Phase 1: divide suffixes into groups
order all suffixes of S into groups: Let S_i and S_j be two suffixes.
Then, group(i) = group(j) if and only if $S[i..\widehat{i}) = S[j..\widehat{j})$.
order the suffix groups by their contexts: Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be two groups,
$i \in \mathcal{G}_1$, $j \in \mathcal{G}_2$. Then, $\mathcal{G}_1 < \mathcal{G}_2$ if and only if $S[i..\widehat{i}) <_{\mathsf{lex}} S[j..\widehat{j})$.
Phase 2: construct suffix array from groups
$\mathsf{SA}[1] \leftarrow n$
for $i = 1$ **up to** n **do**
  for all suffixes S_j with $\widehat{j} = \mathsf{SA}[i]$ **do**
    let sr be the number of suffixes placed in lower groups,
    i.e. $sr := |\{\ s \in [1 \ldots n] \mid \text{group}(s) < \text{group}(j)\ \}|$.
    $\mathsf{SA}[sr + 1] \leftarrow j$
    remove j from its current group and put it in a new group
    placed as immediate predecessor of j's old group.

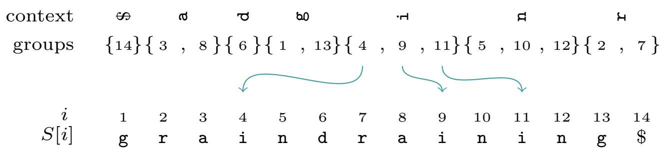


Figure 1 Initial group division for the suffixes of S = `graindraining$`, where links from the group with context `i` to the text are shown. Groups are ordered by their context from left to right.

Because $S_{\widehat{j}} <_{\mathsf{lex}} S_j$ holds by the definition of next lexicographically smaller suffixes, the index $\widehat{j}$ must have been processed by the outer loop of Phase 2 already, and thus, the index j must have been placed to the suffix array correctly, $\mathsf{SA}[i + 1] = j$ holds.

The argumentation shows that the principle works correctly, but there are still a lot of issues remaining. But instead of presenting a more detailed algorithm directly, an introductory example will be presented, to bridge the gap between the sorting principle and the final algorithm.

3.1 Example: Phase 1

Within Phase 1, suffixes have to be divided into groups. More specifically, all suffixes S_i sharing the same prefix $S[i..\widehat{i})$ must belong to the same group, while the groups itself must be sorted by their contexts, see Algorithm 1. To accomplish this task, in an initial step, suffixes are split into groups by their first character. Also, the groups are sorted by their initial context, see Figure 1 for an example.

To obtain the requested group order, all groups are processed in descending order (i.e. from highest to lowest group), repeating the following steps for each group $\mathcal{G}$:

1. For each index $i \in \mathcal{G}$ compute its *prev pointer* prev(i), the previous index placed in a lower group, i.e. $\text{prev}(i) := \max\{\ j \in [1..i] \mid \text{group}(j) < \text{group}(i)\ \}$.
2. Split the set $\mathcal{P} := \{\ \text{prev}(i) \mid i \in \mathcal{G}\ \}$ into subsets $\mathcal{P}_1, \ldots, \mathcal{P}_k$ such that $i, j \in \mathcal{P}_q \Leftrightarrow i, j \in \mathcal{P}$ and group(i) = group(j) for any subset $\mathcal{P}_q$.
3. For each subset $\mathcal{P}_q$, remove the indices of $\mathcal{P}_q$ from their old group and put them to a new group, placed as immediate successor of their old group.

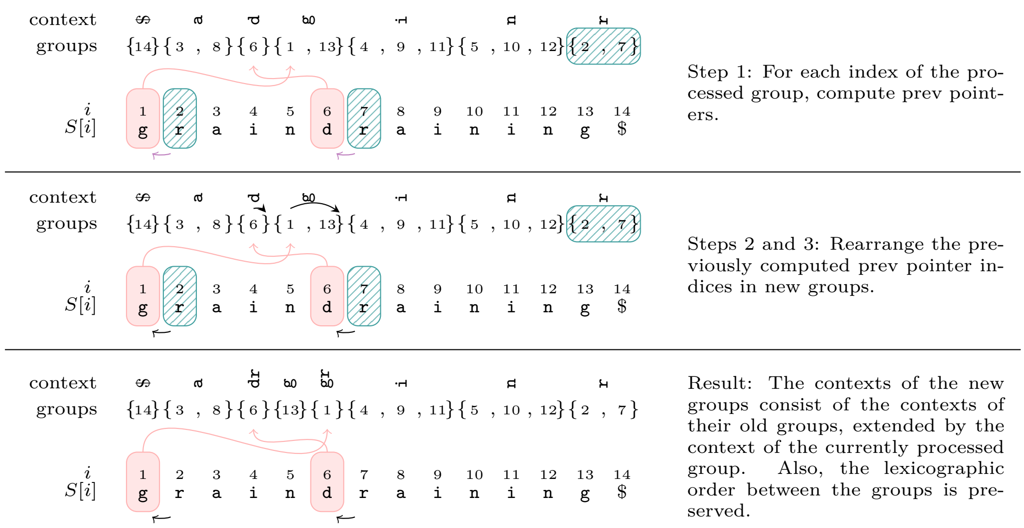


Figure 2 First iteration step of Phase 1 applied to the string $S = \texttt{graindraining}\$$.

Such processing causes an effect quite similar to the *prefix doubling* technique: Each time when indices of a group are removed and collected in a new group (step 3), the context of the new group consists of the context of the old group, extended by the context of the currently processed group, see Figure 2 for an example.

To clarify why context extensions take place, let i be an index and i_{c} be the first index following i such that i is not reachable using the prev pointer chain starting at i_{c}, i.e. $i_{\mathsf{c}} := \min\{\, j \in [i+1..n+1] \mid i \notin \{j, \mathrm{prev}(j), \mathrm{prev}(\mathrm{prev}(j)), \ldots\}\,\}$.[3] As one can show (see [2]), during the processing of groups in Phase 1, $\mathrm{group}(i) = \mathrm{group}(j) \Leftrightarrow S[i..i_{\mathsf{c}}) = S[j..j_{\mathsf{c}})$ holds for two indices i and j, so the string $S[i..i_{\mathsf{c}})$ meets our imagination of group contexts. However, coming back to the above mentioned context extensions, we'll take a closer look onto the steps performed when processing a group. In Step 1, prev pointers are computed. Let i be an index of the processed group, and let $p := \mathrm{prev}(i)$ be its prev pointer. By the definition of a prev pointer (see Step 1), all indices j between p and i ($p < j < i$) are placed in higher groups than p and i.[4] Since groups are processed in decreasing order, for each such index a prev pointer must have been computed already. As p belongs to a lower group than all of those indices, $p \leq \mathrm{prev}(j)$ must hold for all $p < j < i$. Consequently, p is reachable from the prev pointer chains starting at all indices j with $p < j < i$, but as index i had no prev pointer before the current step, $p_{\mathsf{c}} = i$ must hold. Now, after the computation of the prev pointer, p is reachable from all indices up to $i_{\mathsf{c}} - 1$, so the new context of p is $S[p..i_{\mathsf{c}})$. This shows that p's old context was extended by the context of the currently processed group. Consequently, p must be placed into a new group, as performed in Steps 2 and 3.

Another property of the processing is a consistent group order: For any groups $\mathcal{G}_1$ and $\mathcal{G}_2$, $\mathcal{G}_1$ is lower ordered than $\mathcal{G}_2$ if and only if the context of $\mathcal{G}_1$ is lexicographically smaller than the context of $\mathcal{G}_2$. Whenever a new group is created, its context is extended by a lexicographically larger context, so the new group must be placed higher than the old one. Also, since the context of the old group is lexicographically smaller than that of the next

[3] After the initial step $i_{\mathsf{c}} = i + 1$ holds for all indices, because no prev pointers were computed yet.
[4] The special case that groups of indices between p and i are equal to $\mathrm{group}(i)$ will be handled later.

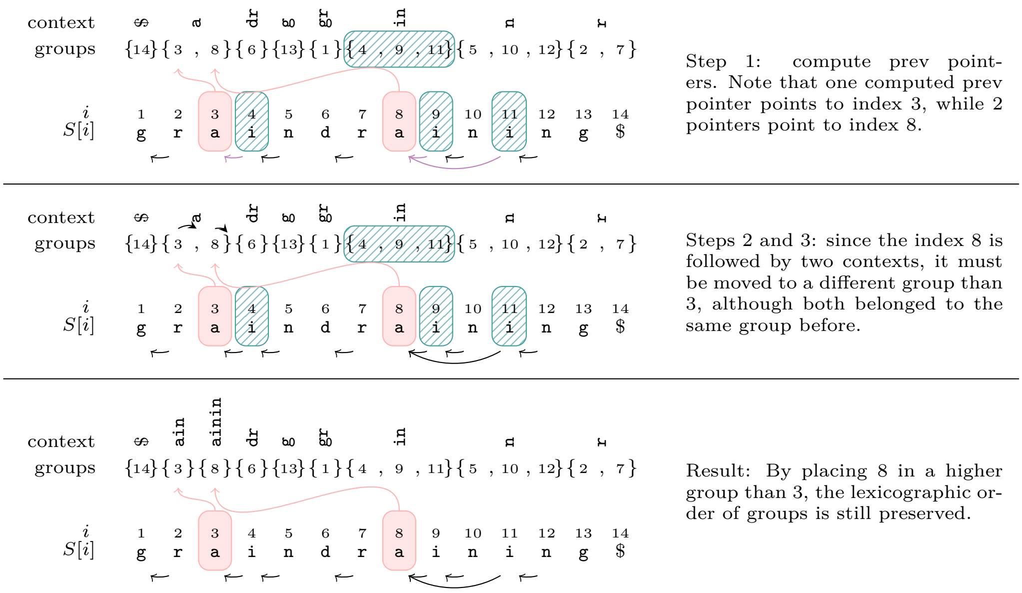


Figure 3 Third iteration step of Phase 1 applied to the string $S =$ `graindraining`\$.

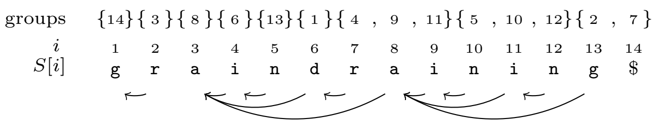


Figure 4 Groups and prev pointers from the string $S =$ `graindraining`\$ after Phase 1.

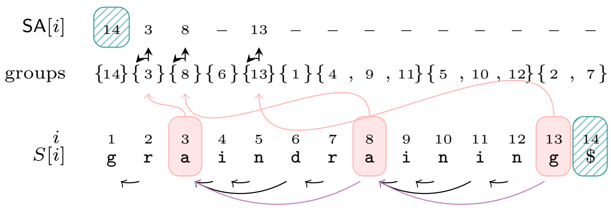
higher group $\widetilde{\mathcal{G}}$, the extended context of the new group is lexicographically smaller than that of $\widetilde{\mathcal{G}}$, so the placement of the new groups in Step 3 preserves the lexicographic order.

Now knowing that context extensions take place, one needs to be aware of one special case to preserve a consistent group order: Think about two indices i and j of the same group such that one prev pointer from an index of the currently processed group points to i, and two prev pointers from the currently processed group point to j. Since context extensions take place, i's context is extended one time, while j's context is extended by two contexts of the currently processed group. Since i and j belong to the same group, the new context of i is lexicographically smaller than that of j. As a consequence, after the extensions, i and j cannot belong to the same group, and must be handled separately as shown in Figure 3. Note that the example considers only two indices with different pointer counts; in general terms, an arbitrary number of indices and pointers must be taken into account.

The result of Phase 1 for our running example can be found in Figure 4. Summarising, the *greedy* group processing from highest to lowest group in conjunction with aspects of *implicit dynamic programming* lead to the desired group division after Phase 1. A formal proof for correctness must be omitted here, but can be found in [2]. Next, we'll take a look at the implementation of the missing part: Phase 2.

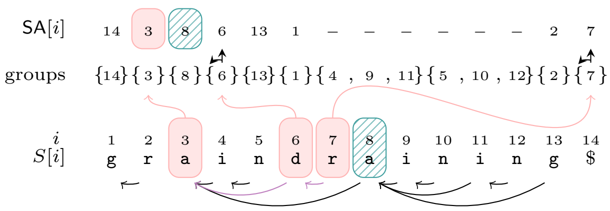
3.2 Example: Phase 2

After dividing suffixes into groups in Phase 1, the purpose of Phase 2 is to compute the suffix array using the group division. During Phase 2, the suffix array is processed in ascending



None of the elements in the prev pointer chain of index $\mathsf{SA}[1]-1=13$ is placed in the suffix array already, so $\widehat{j}=\mathsf{SA}[1]$ holds for each such index. Each index is removed from its current group and placed into a new group as immediate predecessor of its old group. Also, each index is placed into SA, at the position that equals the number of suffixes placed in lower groups.

Figure 5 First iteration step of Phase 2 applied to the string $S=$ `graindraining`\$.



Index 3 is already contained in the suffix array, so only the suffixes S_6 and S_7 are placed into SA, since they are part of the prev pointer chain starting at index $\mathsf{SA}[3]-1=7$.

Figure 6 Third iteration step of Phase 2 applied to the string $S=$ `graindraining`\$.

order. Within the i-th iteration, all indices j with $\widehat{j}=\mathsf{SA}[i]$ are computed. Each such index is removed from its current group, placed into a new group as immediate predecessor of its old group, and stored in the suffix array, see Algorithm 1.

The main issue in implementing this method is to compute indices j with $\widehat{j}=\mathsf{SA}[i]$. As we will see, prev pointers computed in Phase 1 will be very useful for this computation: starting at $j:=\mathsf{SA}[i]-1$, we follow the prev pointer chain $\text{prev}(j),\text{prev}(\text{prev}(j)),\ldots$ until either no more prev pointer exists, or the index under consideration is already contained in the suffix array. The set $\{\, j\in[1\ldots n] \mid \widehat{j}=\mathsf{SA}[i]\,\}$ then consists of exactly those indices visited in the prev pointer chain of $\mathsf{SA}[i]-1$. Examples can be found in Figures 5 and 6, the next purpose is to ensure correctness of this statement.

The first index under consideration is $j:=\mathsf{SA}[i]-1$: if j is not contained in the suffix array already, then by the ascending iteration order of Phase 2, $S_{\mathsf{SA}[i]}<_{\mathsf{lex}}S_j$ must hold. Since S_j is the preceding suffix of $S_{\mathsf{SA}[i]}$, $S_{\mathsf{SA}[i]}$ clearly must be the next lexicographically smaller suffix of S_j. Now, given a suffix S_j with $\widehat{j}=\mathsf{SA}[i]$, the next index k with $\widehat{k}=\mathsf{SA}[i]$ (if existing) can be found by following j's prev pointer, i.e. $k=\text{prev}(j)$. If k is not contained in the suffix array already, $S_{\mathsf{SA}[i]}<_{\mathsf{lex}}S_k$ must hold. Also, since $\text{group}(k)<\text{group}(l)$ holds for all $k<l\leq j$ by the definition of prev pointers, $S_k<_{\mathsf{lex}}S_l$ holds for all $k<l\leq j$ because of the group order of Phase 1. This indeed means that $\widehat{k}\geq\widehat{j}$. Combined with $S_{\mathsf{SA}[i]}<_{\mathsf{lex}}S_k$, $S_{\mathsf{SA}[i]}$ clearly must be the next lexicographically smaller suffix of S_k.

For any index k between j and $\text{prev}(j)$ ($\text{prev}(j)<k<j$) $\text{group}(k)\geq\text{group}(j)$ must hold by the definition of prev pointers. If $\text{group}(k)>\text{group}(j)$, by sorting in Phase 1, $S_k>_{\mathsf{lex}}S_j$ must hold. Because $k<j$, $\widehat{k}\leq j\neq\mathsf{SA}[i]$ holds, so those indices can be skipped. In the special case that $\text{group}(k)=\text{group}(j)$, by Phase 1, $S[k..\widehat{k})=S[j..\widehat{j})$ holds. Since $k<j$ and the contexts are the same, $\widehat{k}<\widehat{j}$ holds, so clearly $\widehat{k}\neq\mathsf{SA}[i]$ must be fulfilled and those indices can be skipped, too.

If an index j is reached that is already contained in the suffix array, we know that it must have been placed into the suffix array in an earlier step. This indeed means that $S_{\widehat{j}}<_{\mathsf{lex}}S_{\mathsf{SA}[i]}$, so j can be skipped. For any further index k in the prev pointer chain of j, an argumentation as above clearly shows that $S_{\widehat{k}}<_{\mathsf{lex}}S_{\mathsf{SA}[i]}$, so those indices can be

Algorithm 2 Suffix array construction of a given nullterminated string S of length n.

Phase 1: divide suffixes into groups
order all suffixes of S into groups according to their first character:
Let S_i and S_j be two suffixes. Then, $\text{group}(i) = \text{group}(j) \Leftrightarrow S[i] = S[j]$.
order the suffix groups: Let $\mathcal{G}_1$ be a suffix group with group context character u, $\mathcal{G}_2$ be a suffix group with group context character v. Then, $\mathcal{G}_1 < \mathcal{G}_2$ if $u < v$.
for each group $\mathcal{G}$ in descending group order **do**
  for each $i \in \mathcal{G}$ **do**
    $\text{prev}(i) \leftarrow \max(\{\ j \in [1 \ldots i] \mid \text{group}(j) < \text{group}(i)\ \} \cup \{0\})$
  let $\mathcal{P}$ be the set of previous suffixes from $\mathcal{G}$,
  $\mathcal{P} := \{\ j \in [1 \ldots n] \mid \text{prev}(i) = j \text{ for any } i \in \mathcal{G}\ \}$.
  split $\mathcal{P}$ into k subsets $\mathcal{P}_1, \ldots, \mathcal{P}_k$ such that a subset $\mathcal{P}_l$ contains suffixes whose number of prev pointers from $\mathcal{G}$ pointing to them is equal to l, i.e. $i \in \mathcal{P}_l \Leftrightarrow |\{\ j \in \mathcal{G} \mid \text{prev}(j) = i\ \}| = l$.
  for $l = k$ **down to** 1 **do**
    split $\mathcal{P}_l$ into m subsets $\mathcal{P}_{l_1}, \ldots, \mathcal{P}_{l_m}$ such that suffixes of same group are gathered in the same subset.
    for $q = 1$ **up to** m **do**
      remove suffixes of $\mathcal{P}_{l_q}$ from their group and put them into a new group placed as immediate successor of their old group.

Phase 2: construct suffix array from groups
$\mathsf{SA}[1] \leftarrow n$
for $i = 1$ **up to** n **do**
  $j \leftarrow \mathsf{SA}[i] - 1$
  while $j \neq 0$ **do**
    let sr be the number of suffixes placed in lower groups, i.e. $sr := |\{\ s \in [1 \ldots n] \mid \text{group}(s) < \text{group}(j)\ \}|$.
    if $\mathsf{SA}[sr + 1] \neq \texttt{nil}$ **then**
      break
    $\mathsf{SA}[sr + 1] \leftarrow j$
    remove j from its current group and put it in a new group placed as immediate predecessor of j's old group.
    $j \leftarrow \text{prev}(j)$

skipped, too. For the remaining indices between this prev pointer chain, we can also use the argumentation above and forget about these indices, too.

We refer to [2] for a formal proof, it must be omitted here for reasons of space. So far, we've seen a running example along with some argumentations for correctness. The missing part is an algorithm along with its runtime analysis, which will be addressed in the next section.

4 Algorithm

The new suffix array construction algorithm including all special cases discussed in the previous section can be found in Algorithm 2.

Now, to verify that the algorithm can be implemented in asymptotic linear time, some technical details about the algorithm will be discussed. First thing that has to be done is to explain a set of needed data structures. Six arrays of size n will be used:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$S[i]$	`g`	`r`	`a`	`i`	`n`	`d`	`r`	`a`	`i`	`n`	`i`	`n`	`g`	`$`
GSIZE$[i]$	1	2	0	1	2	0	3	0	0	3	0	0	2	0
SA$[i]$	14	3	8	6	1	13	4	9	11	5	10	12	2	7
GLINK$[i]$	5	13	2	7	10	4	13	2	7	10	7	10	5	1
ISA$[i]$	5	13	2	7	10	4	14	3	8	11	9	12	6	1

Figure 7 Initial data structure setup after line 2 of Phase 1, applied to the string $S =$ `graindraining$`. Prev pointers are not listed since all entries initially are set to `nil`.

- SA contains suffix starting positions, ordered according to the current group order.
- ISA is the inverse permutation of SA, to be able to detect the position of a suffix in SA.
- GSIZE contains the sizes of all groups. Group sizes are ordered according to the group order, so GSIZE has the same order as SA. GSIZE contains the size of each group once at the beginning of the group, followed by zeros until the beginning of the next group.
- GLINK stores pointers from suffixes to their groups. All entries point to the beginning of a group, at the same position where GSIZE contains the size of the group.
- PREV is used to store prev pointers. All entries initially are set to `nil`.
- PC is used to count prev pointers pointing from $\mathcal{G}$ to $\mathcal{P}$. PC initially is set to zero.

The initial setup of those structures (lines 1 and 2 of Algorithm 2) can be performed in $O(n)$ time using a technique called *bucket sort*. Refer to Figure 7 for an example.

The first problem to be solved is the processing of groups in descending group order, line 3. Therefore, if two variables gs and ge contain the bounds of the current group $\mathcal{G}$ in SA, we get to the preceding group by setting $ge \leftarrow gs - 1$ and $gs \leftarrow \mathsf{GLINK}[\mathsf{SA}[gs-1]]$, and so trivially need $O(n)$ time to iterate over all groups.

For the prev pointer computation in line 5, we observe the following: Each index j between an index i and $\text{prev}(i)$ belongs to a higher or equal group. If j belongs to a higher group, its prev pointer is already computed, and each index between j and $\text{prev}(j)$ belongs to a higher group than that of i. So, to compute the prev pointer of an index i, we start at index $i-1$ and follow prev pointers until an index j belongs to the same or a lower group[5]. If j belongs to a lower group, the prev pointer of i is found; otherwise, if j belongs to the same group and itself has no prev pointer yet, we collect j in a list and repeat the same procedure, thus setting prev pointers of a whole list of indices. This technique is called *pointer jumping* and is well known to require $O(n)$ work totally, since each pointer is used only once for pointer jumping, and overall n pointers are computed. The extra amount of work for the list collection is $O(|\mathcal{G}|)$, and therefore sums up to $O(n)$ in total for Phase 1, since each group is processed only once.

For the computation of the set $\mathcal{P}$ and subsets $\mathcal{P}_1, \ldots, \mathcal{P}_k$[6], (lines 6 to 7) we use the PC-array. After prev pointer computation, for each $i \in \mathcal{G}$, we increment $\mathsf{PC}[\mathsf{PREV}[i]]$. After this loop, $\mathsf{PC}[p]$ contains the count of prev pointers pointing from $\mathcal{G}$ to p. Also note that the set $\mathcal{P}$ easily can be computed during the loop, by adding the index $\text{prev}(i)$ to set $\mathcal{P}$ if $\mathsf{PC}[\text{prev}(i)]$ was zero before the incrementation. Now, while the set $\mathcal{P}$ is not empty, do the following: In the l-th iteration, for each $p \in \mathcal{P}$, decrement $\mathsf{PC}[p]$. If $\mathsf{PC}[p]$ is zero, remove p from $\mathcal{P}$ and add it to set $\mathcal{P}_l$. This way, all sets $\mathcal{P}_1, \ldots, \mathcal{P}_k$ are computed, and all entries of the array PC are set to zero, so it can be reused again. Time results in $O(|\mathcal{G}|)$ per group $\mathcal{G}$, because the

[5] This can be done by comparing $\mathsf{GLINK}[j]$ with gs from the actual group.
[6] The set $\mathcal{P}$ and subsets $\mathcal{P}_1, \ldots, \mathcal{P}_k$ can be implemented as list and list of lists respectively.

Table 2 SACA performance results. Speed[a)] and cache misses[b)] are composed of the arithmetic mean of 10 runs per file for each text corpus.

Text Corpus		divsufsort[12]	SA-IS[13]	KA[9]	DC3[20]	GSACA[1]
Silesia [3]	speed[a)]	15.9 MB/s	17.2 MB/s	8.1 MB/s	2.9 MB/s	4.5 MB/s
(files < 40 MB)	cache misses[b)]	26.5 %	32.7 %	24.2 %	52.0 %	61.2 %
Pizza & Chili [4]	speed[a)]	9.2 MB/s	8.1 MB/s	3.5 MB/s	1.1 MB/s	3.0 MB/s
(files with 200 MB)	cache misses[b)]	49.5 %	74.8 %	55.2 %	86.1 %	79.0 %
Repetitive [5]	speed[a)]	12.5 MB/s	14.2 MB/s	5.3 MB/s	1.7 MB/s	3.5 MB/s
(files > 45 MB)	cache misses[b)]	41.9 %	68.6 %	49.7 %	78.0 %	76.9 %

a) Construction speed: size of input/time to construct SA, in MB/s.
b) Cache miss rate: number of cache misses/number of cache accesses of last–level cache, in %.

number of decrementations in the array PC is identical to the number of additions in the preceding stage, and therefore again, computation requires $O(n)$ work during Phase 1.

The suffix rearrangements from lines 9 to 11 can be performed like the following:

1. For all $p \in \mathcal{P}_l$, decrement GSIZE[GLINK[p]] and exchange p with the index placed at GLINK[p] + GSIZE[GLINK[p]] using SA and ISA. This way, p is moved to the back of its group and 'virtually' removed from it.[7]
2. For all $p \in \mathcal{P}_l$, set GLINK[p] to GLINK[p] + GSIZE[GLINK[p]], so GLINK correctly points to the beginning of the new groups again.
3. For all $p \in \mathcal{P}_l$, increment GSIZE[GLINK[p]], so the sizes of the new groups are correct.

Total work again results in $O(n)$ for Phase 1, for the same reasons as above.

After the processing of a group $\mathcal{G}$ is finished, we also set SA[ge] $\leftarrow gs$ and ISA[i] $= ge$ for all indices $i \in \mathcal{G}$: this serves as a preparation for Phase 2. In Phase 2, to detect if an index j is contained in SA already (line 17), we check if ISA[j] $= 0$ holds; otherwise, sr, the number of suffixes placed in lower groups (line 16), can be computed using ISA and SA. As mentioned above, in Phase 2, ISA entries point to the end of a group. The last index of a group then contains a pointer to the start of the group. If we set $sr \leftarrow$ SA[ISA[j]], increment SA[ISA[j]] and afterwards set SA[sr] $\leftarrow j$ and ISA[j] $\leftarrow 0$, j 'virtually' gets removed from its group, while the group counter points to the next SA - entry.

Now, summing up all work performed, we get $O(n)$ work for Phase 1 as well as for Phase 2, since the inner loop of Phase 2 is executed $n - 1$ times totally, as each suffix has exactly one next lexicographically smaller suffix. There might be smarter ways to implement the algorithm; refer to [2] for other suggestions; however, the point of interest here is that Algorithm 2 can be implemented in a non-recursive way, running in asymptotic linear time.

5 Performance Analyses

All experiments were conducted on a 64 bit Ubuntu 14.04.3 LTS system equipped with two ten-core Intel Xeon processors E5-2680v2 with 2.8 GHz and 128 GB of RAM.

The algorithm described in this paper was named *GSACA* because of its greedy and grouping behaviour. It was compared against common linear-time and state of the art SACAs on text selections of different text corpuses. The benchmark itself is available online [1], results can be found in Table 2.

[7] Note that the additional split of $\mathcal{P}_l$ from line 9 of Algorithm 2 implicitly is performed within this step.

The results clearly show that GSACA cannot keep up with current state of the art SACAs; construction speeds of divsufsort or SA-IS are about 3 to 4 times faster than those of GSACA. Limited performance mainly is owed to cache-unfriendly operations like pointer jumping or suffix rearrangements, causing high cache miss rates and slow construction.

6 Conclusion

We presented the first non-recursive linear-time suffix array construction algorithm. Unfortunately, by comparing its performance with other linear–time SACAs, GSACA must be seen as a late child of the 2003 'epoch of suffix array construction' rather than a state of the art SACA. Nonetheless, the results are quite promising: the algorithm deals a lot with previous smaller and next smaller values, what normally hints to an alternative stack-based approach. This could result in better cache miss rates and speed, but this remains an open problem for the moment. Compared to developmental histories of other SACAs, GSACA is in its infancy, and therefore offers a lot of room for future improvements.

References

1 Uwe Baier. GSACA. `https://github.com/waYne1337/gsaca`. last visited January 2016.
2 Uwe Baier. Linear-time Suffix Sorting-A new approach for suffix array construction. Master's thesis, Ulm University, 2015.
3 Sebastian Deorowicz. Silesia Corpus. `http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia`. last visited January 2016.
4 Paolo Ferragina and Gonzalo Navarro. Pizza & Chili Corpus. `http://pizzachili.dcc.uchile.cl/texts.html`. last visited January 2016.
5 Paolo Ferragina and Gonzalo Navarro. Repetitive Corpus. `http://pizzachili.dcc.uchile.cl/repcorpus.html`. last visited January 2016.
6 Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Breaking a Time-and-Space Barrier in Constructing Full-Text Indices. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '03, pages 251–260, 2003.
7 Juha Kärkkäinen and Peter Sanders. Simple Linear Work Suffix Array Construction. In *Proceedings of the 30th International Conference on Automata, Languages and Programming*, ICALP '03, pages 943–955, 2003.
8 Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time Construction of Suffix Arrays. In *Proceedings of the 14th Annual Conference on Combinatorial Pattern Matching*, CPM '03, pages 186–199, 2003.
9 Pang Ko. Ko–Aluru Algorithm. `https://sites.google.com/site/yuta256/KA.tar.bz2`. last visited January 2016.
10 Pang Ko and Srinivas Aluru. Space Efficient Linear Time Construction of Suffix Arrays. In *Proceedings of the 14th Annual Conference on Combinatorial Pattern Matching*, CPM '03, pages 200–210, 2003.
11 Udi Manber and Gene Myers. Suffix Arrays: A New Method for On-line String Searches. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, pages 319–327, 1990.
12 Yuta Mori. libdivsufsort. `https://github.com/y-256/libdivsufsort`. last visited January 2016.
13 Yuta Mori. sais–lite–2.4.1. `https://sites.google.com/site/yuta256/sais`. last visited January 2016.
14 Yuta Mori. Suffix Array Construction Benchmark. `https://github.com/y-256/libdivsufsort/blob/wiki/SACA_Benchmarks.md`. last visited January 2016.

15 Joong Chae Na. Linear-Time Construction of Compressed Suffix Arrays Using O(N Log N)-bit Working Space for Large Alphabets. In *Proceedings of the 16th Annual Conference on Combinatorial Pattern Matching*, CPM '05, pages 57–67, 2005.

16 Ge Nong. Practical Linear-time O(1)-workspace Suffix Sorting for Constant Alphabets. *ACM Transactions on Information Systems*, 31(3):15:1–15:15, 2013.

17 Ge Nong, Sen Zhang, and Wai Hong Chan. Linear Suffix Array Construction by Almost Pure Induced-Sorting. In *Proceedings of the 2009 Data Compression Conference*, DCC '09, pages 193–202, 2009.

18 Ge Nong, Sen Zhang, and Wai Hong Chan. Linear Time Suffix Array Construction Using D-Critical Substrings. In *Proceedings of the 20th Annual Conference on Combinatorial Pattern Matching*, CPM '09, pages 54–67, 2009.

19 Simon J Puglisi, William F Smyth, and Andrew H Turpin. A Taxonomy of Suffix Array Construction Algorithms. *ACM Computational Survey*, 39(2), 2007.

20 Peter Sanders. DC3 Algorithm. `http://people.mpi-inf.mpg.de/~sanders/programs/suffix/`. last visited January 2016.

21 Peter Weiner. Linear Pattern Matching Algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory*, SWAT '73, pages 1–11, 1973.