10-2018

Optimal in-place suffix sorting

Zhize LI
Singapore Management University, zhizeli@smu.edu.sg

Jian LI

Hongwei HUO

Optimal In-Place Suffix Sorting

Zhize Li[1(✉)], Jian Li[1(✉)], and Hongwei Huo[2]

[1] Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing, China
zz-li14@mails.tsinghua.edu.cn, lijian83@mail.tsinghua.edu.cn
[2] Department of Computer Science, Xidian University, Xi'an, China
hwhuo@mail.xidian.edu.cn

Abstract. The suffix array is a fundamental data structure for many applications that involve string searching and data compression. Designing time/space-efficient suffix array construction algorithms has attracted significant attentions and considerable advances have been made for the past 20 years. We obtain the *first* in-place linear time suffix array construction algorithms that are optimal both in time and space for (read-only) integer alphabets. Our algorithm settles the open problem posed by Franceschini and Muthukrishnan in ICALP 2007. The open problem asked to design in-place algorithms in $o(n \log n)$ time and ultimately, in $O(n)$ time for (read-only) integer alphabets with $|\Sigma| \leq n$. Our result is in fact slightly stronger since we allow $|\Sigma| = O(n)$. Besides, we provide an optimal in-place $O(n \log n)$ time suffix sorting algorithm for read-only general alphabets (i.e., only comparisons are allowed), recovering the result obtained by Franceschini and Muthukrishnan which was an open problem posed by Manzini and Ferragina in ESA 2002.

Keywords: Suffix sorting · Suffix array · In-place

1 Introduction

In SODA 1990, suffix arrays were introduced by Manber and Myers [25] as a space-saving alternative to suffix trees [9,29]. Since then, it has been used as a fundamental data structure for many applications in string processing, data compression, text indexing, information retrieval and computational biology [1,10,12,14,15]. Particularly, the suffix arrays are often used to compute the Burrows-Wheeler transform [5] and Lempel-Ziv factorization [39]. Comparing with suffix trees, suffix arrays use much less space in practice. Abouelhoda et al. [2] showed that any problem which can be computed using suffix trees can also be solved using suffix arrays with the same asymptotic time complexity, which makes suffix arrays very attractive both in theory and in practice. Hence, suffix arrays have been studied extensively over the last 20 years (see e.g., [11,18,19,22,31,33,34]). We refer the readers to the surveys [8,35] for many suffix sorting algorithms.

T. Gagie et al. (Eds.): SPIRE 2018, LNCS 11147, pp. 268–284, 2018.
https://doi.org/10.1007/978-3-030-00479-8_22

In 1990, Manber and Myers [25] obtained the first $O(n \log n)$ time suffix sorting algorithm over general alphabets. In 2003, Ko and Aluru [22], Kärkkäinen and Sanders [18] and Kim et al. [21] independently obtained the first linear time algorithm for suffix sorting over integer alphabets. Clearly, these algorithms are optimal in terms of asymptotic time complexity. However, in many applications, the computational bottleneck is the *space* as we need the space-saving suffix arrays instead of suffix trees, and significant efforts have been made in developing *lightweight* (in terms of space usage) suffix sorting algorithms for the last decade (see e.g., [4,11,13,22,26,28,31–34]). In particular, the ultimate goal in this line of work is to obtain *in-place algorithms* (i.e., $O(1)$ additional space), which are also asymptotically optimal in time.

1.1 Problem Setting

Problem: Given a string $T = T[0 \dots n-1]$ with n characters, we need to construct the *suffix array* (SA) which contains the *indices* of all sorted suffixes of T (see Definition 1 for the formal definition of SA).

Here, we consider the following two popular settings. We measure the space usage of an algorithm in the unit of *words* same as [11,31]. A word contains $\lceil \log n \rceil$ bits. One standard arithmetic or bitwise boolean operation on word-sized operands costs $O(1)$ time.

1. Read-only integer alphabets: The input string T is *read-only*. Each $T[i] \in [1, |\Sigma|]$, where $|\Sigma| = O(n)$. Note that the constant alphabets (e.g., ASCII code) is a special case of integer alphabets, and the read-only integer alphabets is commonly used in practice.
2. Read-only general alphabets: The input string T is read-only and the only operations allowed on the characters of T are comparisons. Each comparison takes $O(1)$ time. We *cannot* write the input space, make bit operations, even copy an input character $T[i]$ to the work space. Clearly, $\Omega(n \log n)$ time is a lower bound for suffix sorting in this case, as it generalizes comparison-based sorting.

The *workspace* used by an algorithm is the total space needed by the algorithm, excluding the space required by the input string T and the output suffix array SA. An algorithm which uses $O(1)$ words workspace to construct SA is called an *in-place* algorithm. See Tables 1[1] and 2 for existing and new results.

1.2 Related Work and Our Contributions

Read-Only Integer Alphabets. In ICALP 2007, Franceschini and Muthukrishnan [11] posed an open problem for designing an in-place algorithm that takes $o(n \log n)$ time or ultimately $O(n)$ time for (read-only) integer alphabets with

[1] Some previous algorithms state the space usages in terms of bits. We convert them into words.

Table 1. Time and workspace of suffix sorting algorithms for read-only integer alphabets Σ

Time	Workspace (words)	Algorithms
$O(n^2 \log n)$	$cn + O(1)$ $c < 1$	[26–28]
$O(n^2 \log n)$	$\|\Sigma\| + O(1)$	[16]
$O(n^2)$	$O(n)$	[38]
$O(n \log^2 n)$	$O(n)$	[36]
$O(n \log n)$	$O(n)$	[23,25]
$O(vn)$	$O(n/\sqrt{v})$ $v \in [1, \sqrt{n}]$	[19]
$O(n\sqrt{\|\Sigma\| \log(n/\|\Sigma\|)})$	$O(n)$	[3]
$O(n \log \log n)$	$O(n)$	[20]
$O(n \log \log \|\Sigma\|)$	$O(n \log \|\Sigma\| / \log n)$	[13]
$O(n \log \|\Sigma\|)$	$\|\Sigma\| + O(1)$	[32]
$O(n)$	$O(n)$	[18,19,21,22]
$O(n)$	$n + n/\log n + O(1)$	[33,34]
$O(n)$	$\|\Sigma\| + O(1)$	[31]
$O(n)$	$O(1)$	This paper

$|\Sigma| \leq n$ (in fact, they did not specify whether the input string T is read-only or not). The current best result along this line is provided by Nong [31], which used $|\Sigma|$ words workspace (Nong's algorithm is in-place if $|\Sigma| = O(1)$, i.e., constant alphabets). Note that in the worst case $|\Sigma|$ can be as large as $O(n)$. We list several previous results and our new result in Table 1.

In this paper, we settle down this open problem by providing the first optimal linear time in-place algorithm, as in the following theorem. Note that our result is in fact slightly stronger since we allow $|\Sigma| = O(n)$ instead of $|\Sigma| \leq n$ mentioned in the open problem [11].

Theorem 1 (Main Theorem). *There is an in-place linear time algorithm for suffix sorting over integer alphabets, even if the input string T is read-only and the size of the alphabet $|\Sigma|$ is $O(n)$.*

Read-Only General Alphabets. Now, we consider the case where the only operations allowed on the characters of string T (read-only) are comparisons. See Table 2 for an overview of the results. In 2002, Manzini and Ferragina [28] posed an open problem, which asked whether there exists an $O(n \log n)$ time algorithm using $o(n)$ workspace. In 2007, Franceschini and Muthukrishnan [11] obtained the first in-place algorithm that runs in optimal $O(n \log n)$ time. Their conference paper is somewhat complicated and densely-argued.

We also give an optimal in-place algorithm which achieves the same result, as in the following theorem. In addition, our algorithm does not make any bit

Table 2. Time and workspace of suffix sorting algorithms for read-only general alphabets

Time	Workspace (words)	Algorithms
$O(n \log n)$	$O(n)$	[23,25]
$O(vn + n \log n)$	$O(v + n/\sqrt{v})$ $v \in [2, n]$	[4]
$O(vn + n \log n)$	$O(n/\sqrt{v})$ $v \in [1, \sqrt{n}]$	[19]
$O(n \log n)$	$O(1)$	[11]
$O(n \log n)$	$O(1)$	This paper

operations while theirs uses bit operations heavily. Our algorithm is also arguably simpler.

Theorem 2. *There is an in-place $O(n \log n)$ time algorithm for suffix sorting over general alphabets, even if the input string T is read-only and only comparisons between characters are allowed.*

1.3 Difficulties and Our Approach

Difficulties: Typically, the suffix sorting algorithms are recursive algorithms. The size of the recursive (reduced) sub-problem is usually less than half of the current problem. See e.g., [11,19,22,31,33–35]. However, all previous algorithms require extra arrays, e.g., *bucket array* (which needs $|\Sigma|$ words at the top recursive level and $n/2$ words at the deep recursive levels), *type array* (which needs $n/\log n$ words) and/or other auxiliary arrays (which need up to $O(n)$ words), to construct the reduced problems and use the results of the reduced problems to sort the original suffixes.[2]

In particular, Nong et al. [33] made a breakthrough by providing the SA-IS algorithm which only required one bucket array (which needs $\max\{|\Sigma|, n/2\}$ words) and one type array ($n/\log n$ words). Note that the bucket array and type array are reused for each recursive level.

Currently, the best result was provided by Nong [31]. However, Nong's algorithm still requires the bucket array for the top recursive level, but not for the deeper levels. Hence, it needs $|\Sigma|$ words instead of $\max\{|\Sigma|, n/2\}$ words. Note that $|\Sigma|$ can be $O(n)$ in the worst case for integer alphabets. For the type array, Nong used this bucket array to indicate the type information at the top recursive level. For the deeper levels, Nong removed the type array.

Thus, *the main technical difficulty is to remove the workspace for the bucket array at the top recursive level since there is no extra space to use.* Note that it is non-trivial since T is read-only and SA needs to store the final order of all suffixes. Besides, the previous sorting steps or tricks may not work if one removes the bucket array. For example, Nong [31] used the bucket array to indicate the

[2] The definitions of bucket array and type array can be found in Sect. 2.

type information. If the bucket array was removed, one would need the type array.

Our Approach: We briefly describe our optimal in-place linear time suffix sorting algorithms that overcome these difficulties. We provide an *interior counter trick* which can implicitly represent the dynamic *LF/RF-entry* information (see Sect. 2 for the definition) in SA. Besides, we provide a *pointer data structure* which can represent the bucket heads/tails in SA. Combining these two techniques, we can remove the workspace needed by the bucket array entirely. Note that it is non-trivial for the top recursive level which is the most difficult part, since the pointer data structure needs *nonconstant* workspace and we only have $O(1)$ extra workspace. As a result, we divide the sorting step into two stages to address this issue. In order to remove the type array, we provide some useful properties and observations which allows us to retrieve the type information efficiently. For the general alphabets case, we provide simple sorting steps and extend the interior counter trick to obtain an optimal in-place $O(n \log n)$ time suffix sorting algorithm.

Organization: The remaining of the paper is organized as follows. Section 2 covers the preliminary knowledge. In Sect. 3, we describe the framework and the details of our optimal in-place suffix sorting algorithm for the read-only integer alphabets. Finally, we conclude in Sect. 4. For the read-only general alphabets, we defer the details of our optimal in-place algorithm to the full version of this paper [24].

2 Preliminaries

Given a string $T = T[0 \dots n-1]$ with n characters, the suffixes of T are $T[i \dots n-1]$ for all $i \in [0, n-1]$, where $T[i \dots j]$ denotes the substring $T[i]T[i+1] \dots T[j]$ in T. To simplify the argument, we assume that the final character $T[n-1]$ is a sentinel which is lexicographically smaller than any other characters in Σ. Without loss of generality, we assume that $T[n-1] = 0$.[3] Any two suffixes in T must be different since their lengths are different, and their lexicographical order can be determined by comparing their characters one by one until we see a difference due to the existence of the sentinel.

Definition 1. *The suffix array* SA *contains the indices of all suffixes of* T *which are sorted in lexicographical order, i.e.,* $\mathsf{suf}(\mathsf{SA}[i]) < \mathsf{suf}(\mathsf{SA}[j])$ *for all* $i < j$, *where* $\mathsf{suf}(i)$ *denotes the suffix* $T[i \dots n-1]$.

For example, if $T =$ "1220", then all suffixes are $\{1220, 220, 20, 0\}$ and $\mathsf{SA} = [3, 0, 2, 1]$. Note that SA always uses n words no matter what the alphabets Σ are, since it contains the permutation of $\{0, \dots, n-1\}$, where n is the length of T.

A suffix $\mathsf{suf}(i)$ is said to be *S-suffix* (S-type suffix) if $\mathsf{suf}(i) < \mathsf{suf}(i+1)$. Otherwise, it is *L-suffix* (L-type suffix) [22]. The last suffix $\mathsf{suf}(n-1)$ containing

[3] Some previous papers use \$ to denote the sentinel.

only the single character 0 (the sentinel) is defined to be an S-suffix. Equivalently, the $\mathsf{suf}(i)$ is S-suffix if and only if (1) $i = n-1$; or (2) $T[i] < T[i+1]$; or (3) $T[i] = T[i+1]$ and $\mathsf{suf}(i+1)$ is S-suffix. Obviously, the types can be computed by a linear scan of T (from $T[n-1]$ to $T[0]$). We further define the type of a character $T[i]$ to *S-type* (or *L-type* resp.) if $\mathsf{suf}(i)$ is S-suffix (or L-suffix resp.). For the same example, the types are "*SLLS*".

A suffix $\mathsf{suf}(i)$ is called an *LMS-suffix* (Leftmost S-type) if $T[i]$ is S-type and $T[i-1]$ is L-type, for $i \geq 1$ [33]. Similarly, a character $T[i]$ is called *LMS-character* if $\mathsf{suf}(i)$ is LMS-suffix. A substring $T[i \dots j]$ is called an *LMS-substring* if both $T[i]$ and $T[j]$ are LMS-characters, and there is no other LMS-characters between them, or $i = j = n-1$ (the single sentinel). We also define the *LML-suffix* (Leftmost L-type), *LML-character* and *LML-substring* similarly.

Obviously, the indices of all suffixes, which begin with the same character, must appear consecutively in SA. We denote a subarray in SA for these suffixes with the same beginning character as a *bucket*, where the *head* and the *tail* of a bucket refer to the first and the last index of the bucket in SA respectively. Moreover, we define the first common character as its *bucket character*. For the same example $\mathsf{SA} = [3, 0, 2, 1]$, the buckets are $\{\mathsf{SA}[0], \mathsf{SA}[1], \mathsf{SA}[2, 3]\}$ and the bucket characters are $0, 1, 2$, respectively. If the bucket character is $T[i]$, we refer to the bucket as bucket $T[i]$. The bucket head and tail of bucket 2 is 2 and 3, respectively. Note that S-suffixes always appear after the L-suffixes in any bucket, i.e., if an S-suffix and an L-suffix begin with the same character, the L-suffix is always smaller than the S-suffix.

Induced Sorting: The *induced sorting* technique, developed by Ko and Aluru [22], is responsible for many recent advances of suffix sorting algorithms [11,31, 33–35], and is also crucial to us. It can be used to induce the lexicographical order of L-suffixes from the sorted S-suffixes. Now, we briefly introduce the standard induced sorting technique which needs the *bucket array* and *type array* explicitly. The bucket array contains $|\Sigma|$ integers and each denotes the position of a bucket head/tail (depending on induced sorting the L-suffixes or S-suffixes) in SA. The type array contains n bits and each entry denotes an L/S-type information for T (i.e., 0 for L-type and 1 for S-type).

Inducing the Order of L-suffixes from the Sorted S-suffixes: Assume that all indices of the sorted S-suffixes are already in their correct positions in SA (i.e., in the tail of their corresponding buckets in SA). Now, we define some new notations (e.g., *LF/RF-entry*) to simplify the representation. We scan SA from left to right (i.e., from $\mathsf{SA}[0]$ to $\mathsf{SA}[n-1]$). We maintain an *LF-pointer* (leftmost free pointer) for each bucket which points to the leftmost free entry (called the *LF-entry*) of the bucket. The LF-pointers initially point to the head of their corresponding buckets. When we scan $\mathsf{SA}[i]$, let $j = \mathsf{SA}[i] - 1$. If $\mathsf{suf}(j)$ is an L-suffix (indicated by the type array), we place the index of $\mathsf{suf}(j)$ (i.e., j) into the LF-entry of bucket $T[j]$, and then let the LF-pointer of this bucket $T[j]$ point to the next free entry. The LF-pointers are maintained in the bucket array. If $\mathsf{suf}(j)$ is an S-suffix, we do nothing (since all S-suffixes are already sorted in the correct positions). We give a running example in our full version [24].

Sorting all S-suffixes from the sorted L-suffixes is completely symmetrical: we scan SA from right to left, maintaining an *RF-pointer* (rightmost free pointer) for each bucket which points to the *RF-entry* (rightmost free entry) of the bucket.

The idea of induced sorting is that the lexicographical order between $\mathsf{suf}(i)$ and $\mathsf{suf}(j)$ is decided by the order of $\mathsf{suf}(i+1)$ and $\mathsf{suf}(j+1)$ if $\mathsf{suf}(i)$ and $\mathsf{suf}(j)$ are in the same bucket (i.e., $T[i] = T[j]$). We only need to specify the correct order of these L-suffixes in the same buckets since we always place the L-suffixes in their corresponding buckets. Consider two L-suffixes $\mathsf{suf}(i)$ and $\mathsf{suf}(j)$ in the same bucket. We have $\mathsf{suf}(i+1) < \mathsf{suf}(i)$ and $\mathsf{suf}(j+1) < \mathsf{suf}(j)$ by the definition of L-suffix. Since we scan SA from left to right, $\mathsf{suf}(i+1)$ and $\mathsf{suf}(j+1)$ must appear earlier than $\mathsf{suf}(i)$ and $\mathsf{suf}(j)$. Hence the correctness of induced sorting is not hard to prove by induction.

Actually, Nong et al. [33] observed that one can sort all L-suffixes from the sorted LMS-suffixes (instead of S-suffixes). Roughly speaking, the idea is that in the induced sorting, only LMS-suffixes are useful for sorting L-suffixes. We also provide a running example in the full version [24]. They also showed that one can use the same induced sorting step to sort all LMS-substrings from the sorted LMS-characters of T.

Note that in this preliminary section, the induced sorting steps are not *in-place* since they require explicit storage for the bucket and type arrays.

3 Suffix Sorting for Read-Only Integer Alphabets

In this section, we provide an *interior counter trick* which can implicitly represent the dynamic LF/RF-entry information in SA. Besides, we provide a *pointer data structure* which can represent the bucket heads/tails in SA. Combining these two techniques, we can remove the workspace needed by the bucket array entirely. To address the issue of the hardest part (i.e. the top recursive level), we divide the sorting step into two stages. For removing the type array, we give some useful properties and observations between string T and SA to obtain the L/S-type information.

3.1 Framework

First, we define some notations. Let n_L and n_S denote the number of L-suffixes and S-suffixes, respectively. Let n_1 denote the length of the reduced problem T_1, i.e., n_1 equals to the number of LMS-suffixes (Case 1) or LML-suffixes (Case 2). Note that the number of LMS-characters, LMS-suffixes, and LMS-substrings are the same. Now, we describe the framework of our algorithm as follows:

1. If $n_L \leq n_S$ (i.e., the number of L-suffixes is no larger than that of S-suffixes), then:
   (1) (Sect. 3.2) Sort all LMS-characters of T.
   We use counting sort to sort all LMS-characters of T in $\mathsf{SA}[n-n_1 \ldots n-1]$. In the counting sort step, we use $\mathsf{SA}[0 \ldots n/2]$ as the temporary space (counting array). After this step, all indices of the sorted LMS-characters are stored in $\mathsf{SA}[n-n_1 \ldots n-1]$.

(2) (Sect. 3.4) Induced sort all LMS-substrings from the sorted LMS-characters.
This induced-sorting step is the same as Step (4) below where we induced-sort all suffixes from the sorted LMS-suffixes. Thus, we only describe the details of this step in Sect. 3.4. After this step, all indices of the sorted LMS-substrings are stored in $\mathsf{SA}[n - n_1 \dots n - 1]$.

(3) (Sect. 3.3) Construct and solve the reduced problem T_1 from the sorted LMS-substrings.
We construct the reduced problem T_1 using the ranks of all sorted LMS-substrings which are stored in $\mathsf{SA}[n-n_1 \dots n-1]$, where the ranks of LMS-substrings correspond to the lexicographical order of the sorted LMS-substrings. Then we get the reduced problem T_1 in $\mathsf{SA}[0 \dots n_1 - 1]$ and solve T_1 recursively to obtain the sorted LMS-suffixes. In the recursive step, we use $\mathsf{SA}_1 = \mathsf{SA}[n-n_1 \dots n-1]$ as the output space for T_1. After this step, all indices of the sorted LMS-suffixes are stored in $\mathsf{SA}[n-n_1 \dots n-1]$.

(4) (Sect. 3.4) Induced sort all suffixes of T from the sorted LMS-suffixes (T_1).
We induced-sort all suffixes of T from the sorted LMS-suffixes which are stored in $\mathsf{SA}[n - n_1 \dots n - 1]$. Note that the in-place implementation of this induced sorting step is the main technical part of our optimal in-place algorithm. As we discussed before, we develop the interior counter trick and the pointer data structure, and then divide this sorting step into two stages to remove the workspace. After this step, all indices of the suffixes of T are sorted and stored in $\mathsf{SA}[0 \dots n - 1]$.

2. Otherwise, execute the above steps switching the role of LMS with LML.

Without loss of generality, we assume that $n_L \leq n_S$. Note that we compare the number of L-suffixes and S-suffix at the beginning since we need half of the space of SA to construct our pointer data structure for induced-sorting the L-suffixes (from the sorted LMS-suffixes) and S-suffixes (from the sorted L-suffixes) in Step (4). Note that the empty space is enough since the number of LMS-suffixes (i.e., n_1) and L-suffixes (i.e., n_L) both are less than or equal to $n/2$, where $n_1 \leq n/2$ since any two LMS-characters are not adjacent by the definition of LMS-characters, and $n_L \leq n/2$ since $n_L \leq n_S$. Note that for previous algorithms (e.g., [31,33]), they do not need the comparison at the beginning since they use the bucket array (which needs $|\Sigma|$ words workspace) in the induced sorting step (i.e. Step (4)). Here, we construct the pointer data structure and combine our interior counter trick to remove the bucket array.

Now, we describe the details of our in-place algorithm in the following sections.

3.2 Sort All LMS-Characters of T

In this section, we sort all LMS-characters of T and place their indices in $\mathsf{SA}[n - n_1 \dots n - 1]$. Recall that n_1 denotes the number of LMS-characters.

Now, we describe the details. Since $|\Sigma| = O(n)$, we can assume that $|\Sigma| \leq dn$ for some constant d. We divide the LMS-characters of T into $2d$ partitions and

sort each partition one by one. The partition i contains the LMS-characters which belong to $\left[\frac{i|\Sigma|}{2d}+1, \frac{(i+1)|\Sigma|}{2d}\right]$, for $0 \leq i < 2d$. We use m_i to denote the number of LMS-characters in partition i. Then for each partition i, we use the standard *counting sort* (see e.g., [7, Chap. 8]) to sort these m_i LMS-characters (the LMS-characters can be identified by scanning T once from right to left). Concretely, we use $\mathsf{SA}[0 \ldots n/2]$ as the temporary counting array, and use $\mathsf{SA}[n/2+\sum_{j=0}^{i-1} m_j + 1 \ldots n/2+\sum_{j=0}^{i} m_j]$ as the output array. After this counting sort step, the indices of these m_i sorted LMS-characters have been placed in $\mathsf{SA}[n/2+\sum_{j=0}^{i-1} m_j + 1 \ldots n/2+\sum_{j=0}^{i} m_j]$.

Note that we can use the counting sort step for each partition. Because the gap of each partition is $\frac{|\Sigma|}{2d} \leq \frac{dn}{2d} = \frac{n}{2}$, the space of $\mathsf{SA}[0 \ldots n/2]$ is enough for the temporary counting array (its size equals to the gap) of counting sort step. It is not hard to see that the sorting step takes $O(n)$ time and uses $O(1)$ workspace since we only make $2d$ times of counting sort steps (each step takes linear time).

After sorting all $2d$ partitions, all indices of the sorted LMS-characters are placed in $\mathsf{SA}[n/2+1, n/2+\sum_{j=0}^{2d-1} m_j]$ (i.e., $\mathsf{SA}[n/2+1, n/2+n_1]$). Then we move them to $\mathsf{SA}[n-n_1 \ldots n-1]$, which can be easily done in linear time and $O(1)$ workspace.

3.3 Sort All LMS-Suffixes of T by Solving the Reduced Problem T_1

Construct the Reduced Problem T_1: We construct the reduced problem T_1 using the ranks of all sorted LMS-substrings which are stored in $\mathsf{SA}[n-n_1 \ldots n-1]$ from the Step (2) (see the framework in Sect. 3.1), where the ranks of LMS-substrings are corresponding to the lexicographical order of the sorted LMS-substrings. Note that this construction step is not difficult and similar to the previous algorithms (e.g., [31,33]).

Now, we spell out the details for this step. Initially, all LMS-substrings are sorted in $\mathsf{SA}[n-n_1 \ldots n-1]$. First, we let the rank of the smallest LMS-substring (i.e., the LMS-substring which begins from index $\mathsf{SA}[n-n_1]$) be 0 (it must be the sentinel). Then, we scan $\mathsf{SA}[n-n_1+1 \ldots n-1]$ from left to right to compute the rank for each LMS-substring. When scanning $\mathsf{SA}[i]$, we compare the LMS-substring corresponding to $\mathsf{SA}[i]$ and that corresponding to $\mathsf{SA}[i-1]$. If they are the same, $\mathsf{SA}[i]$ gets the same rank as $\mathsf{SA}[i-1]$. Otherwise, the rank of $\mathsf{SA}[i]$ is the rank of $\mathsf{SA}[i-1]$ plus 1. Since we have no extra space, we need to store the ranks in SA as well. In particular, the rank of $\mathsf{SA}[i]$ is stored in $\mathsf{SA}[\lfloor \frac{\mathsf{SA}[i]}{2} \rfloor]$. There is no conflict since any two LMS-characters are not adjacent. Finally, we shift nonempty entries in $\mathsf{SA}[0 \ldots n-n_1-1]$ to the head of SA, so that the ranks occupy a consecutive segment of the space. Now, we have obtained the reduced problem T_1 which is stored in $\mathsf{SA}[0 \ldots n_1-1]$. In other words, $\mathsf{SA}[i]$ ($i \in [0, n_1-1]$) stores the new name of the i-th LMS-substring with respect to its appearance in the input string T. An example of this step can be found in our full version [24].

Now, we have the following lemma.

Lemma 1. *T_1 can be constructed and stored in $\mathsf{SA}[0 \dots n_1 - 1]$ using $O(n)$ time and $O(1)$ workspace.*

The proof easily follows from the following observation, i.e., the whole comparison process takes $O(n)$ time because the total length of all LMS-substrings (each of them is identified by this observation) is less than $2n$.

Observation 1. *For any index i of T, let $j \in [i+1, n-1]$ be the smallest index such that $T[j] < T[j+1]$ (So $T[j]$ is S-type). Furthermore let $k \in [i+1, j]$ be the smallest index such that $T[l] = T[j]$ for any $k \le l \le j$. Then $T[k]$ is the first S-type character after index i. Moreover, all characters between $T[i]$ and $T[k]$ are L-type, and characters between $T[k]$ and $T[j]$ are S-type.*

Solve T_1 Recursively: Now, we sort all LMS-suffixes by solving T_1 recursively and place their indices in the tail of SA (i.e. $\mathsf{SA}[n - n_1 \dots n - 1]$). This step is carried out as follows:

1. We first solve T_1 recursively. Recall that T_1 is stored in $\mathsf{SA}[0 \dots n_1 - 1]$. We define SA_1 to be $\mathsf{SA}[n - n_1 \dots n - 1]$ and use SA_1 to store the output of the subproblem T_1.
2. Now, we put all indices of LMS-suffixes in SA. First we move SA_1 to $\mathsf{SA}[0 \dots n_1 - 1]$ (i.e., move $\mathsf{SA}[n - n_1 \dots n - 1]$ to $\mathsf{SA}[0 \dots n_1 - 1]$). Then we scan T from right to left. For every LMS-character $T[i]$, place i (i.e., index of $\mathsf{suf}(i)$) in the tail of SA.
3. For notational convenience, we define $\mathsf{LMS}[0 \dots n_1] \triangleq \mathsf{SA}[n - n_1 \dots n - 1]$. Now, we obtain the sorted order of all LMS-suffixes of the original string T by letting $\mathsf{SA}[i] = \mathsf{LMS}[\mathsf{SA}[i]]$ for all $i \in [0, n_1 - 1]$.
4. Finally, we finish this step by moving $\mathsf{SA}[0 \dots n_1 - 1]$ to $\mathsf{SA}[n - n_1 \dots n - 1]$. Now, all indices of the sorted LMS-suffixes are stored in $\mathsf{SA}[n - n_1 \dots n - 1]$.

Lemma 2. *All LMS-suffixes can be sorted by solving the reduced problem T_1 recursively and placed in the tail of SA using $O(n)$ time and $O(1)$ workspace.*

Proof. The time and space used in this step are easy to verify.[4] We only show the correctness of this step. Each character of T_1 corresponds to an LMS-substring of T and this character is the rank of the corresponding sorted LMS-substring. Hence, the lexicographical order of LMS-suffixes of T is the same as the order of suffixes in T_1.

3.4 Induced-Sort All Suffixes of T from the Sorted LMS-Suffixes

In this section, we show how to *in-place* induced-sort all suffixes from the sorted LMS-suffixes which have been placed in $\mathsf{SA}[n-n_1 \dots n-1]$ from the previous step (see Lemma 2). Let $\mathsf{SA}_L = \mathsf{SA}[0 \dots n_L - 1]$ and $\mathsf{SA}_S = \mathsf{SA}[n_L \dots n - 1]$. Recall

[4] If one worries the $O(\log n)$ workspace in the recursion, one can use the highest bits in SA (i.e., n bits) to store them since the size of the reduced sub-problem is no larger than $n/2$.

that n_S and n_L denote the number of S-suffixes and L-suffixes, respectively. Also note that $n_L + n_S = n$. First, we sort all n_L L-suffixes from the sorted LMS-suffixes which are stored in $\mathsf{SA}[n - n_1 \dots n - 1]$ and store the sorted L-suffixes in SA_L. Then, we sort all n_S S-suffixes from the sorted L-suffixes and store the sorted S-suffixes in SA_S. Note that sorting the L-suffixes from the sorted LMS-suffixes is totally symmetrical as sorting the S-suffixes from the sorted L-suffixes, as stated in Sect. 2. Thus, we only need to show the details of how to sort all n_S S-suffixes from the sorted L-suffixes which has already been stored in SA_L, and then store the sorted S-suffixes in SA_S.

We briefly recall the original (not in-place) induced sorting step here. We scan SA from right to left (i.e., from $\mathsf{SA}[n-1]$ to $\mathsf{SA}[0]$). When we scan $\mathsf{SA}[i]$, let $j = \mathsf{SA}[i] - 1$. If $\mathsf{suf}(j)$ is an S-suffix (indicated by the type array), we place the index of $\mathsf{suf}(j)$ (i.e. j) into the RF-entry of bucket $T[j]$, and then let the RF-pointer of this bucket $T[j]$ point to the next free entry. If $T[j]$ is L-type, we do nothing (since all L-suffixes are already sorted). The RF-pointers are maintained by the bucket array.

Inducing the Order of S-suffixes from the Sorted L-suffixes: In order to obtain the in-place algorithm, we develop the interior counter trick and the pointer data structure to remove the workspace needed by the bucket array and type array in the induced sorting step. Briefly speaking, the purpose of the pointer data structure is to indicate the bucket tails of S-suffixes, and the purpose of the interior counter trick is to maintain the RF-pointers of the buckets dynamically. Thus, for a query of RF-entry for $\mathsf{suf}(j)$ in bucket $T[j]$, we know the tail of the bucket $T[j]$ from the pointer data structure in constant time (Lemma 6), then we use the interior counter trick to indicate the RF-entry in this bucket (Lemma 3). For removing the type array, we use the Lemma 4 to identify the L/S-suffixes in the induced sorting step.

Now, we describe the details step by step. First, we introduce our interior counter trick assuming that the tail of the bucket of any S-suffix is known (which is indicated by the Lemma 6).

Interior Counter Trick: Note that the buckets of the S-suffixes we discussed in this section are in $\mathsf{SA}_S = \mathsf{SA}[n_L \dots n - 1]$, since we already have placed the sorted L-suffixes in $\mathsf{SA}_L = \mathsf{SA}[0 \dots n_L - 1]$. Thus, we only need to sort all S-suffixes to their corresponding buckets in SA_S and the buckets only contains S-suffixes now.

Here we only describe the details of interior counter trick for one bucket since other buckets are the same. Recall that we assume that the tail of the bucket of any S-suffix is known (Lemma 6). Thus, to simplify the representation, we assume the bucket from index 0 to index $m - 1$ of SA_S, where m is the size of this bucket (i.e. the number of S-suffixes in this bucket is m). We only describe the case where $m > 3$ since other cases with $m \leq 3$ are similar and simpler. We

define five special symbols $\mathsf{B_H}$ (head of the bucket), $\mathsf{B_T}$ (tail of the bucket), E (Empty), $\mathsf{R_1}$ (one remaining S-suffix) and $\mathsf{R_2}$ (two remaining S-suffixes)[5].

First, we use three special symbols to initialize this bucket, i.e., let $\mathsf{SA}_S[0] = \mathsf{B_H}$, $\mathsf{SA}_S[m-2] = \mathsf{E}$ and $\mathsf{SA}_S[m-1] = \mathsf{B_T}$. Let S_i denote the index of the i-th S-suffix which needs to be placed into the RF-entry of this bucket. Now, we describe how to place the indices of these m S-suffixes into the RF-entry of this bucket one by one. We distinguish the following four cases (To demonstrate these four cases more clearly, we also provide a demonstration in our full version [24].):

(1) If $\mathsf{SA}_S[m-1] = \mathsf{B_T}$, and $\mathsf{SA}_S[m-2] = \mathsf{E}$ or $\mathsf{SA}_S[m - \mathsf{SA}_S[m-2] - 3] \neq \mathsf{B_H}$: In this case, we place the index of the current S-suffix (i.e., S_i) into the RF-entry of this bucket, where $1 \leq i \leq m-3$. Concretely, we know the position of the tail of this bucket in SA_S, i.e., $m-1$ according to the assumption. Then, we use $\mathsf{SA}_S[m-2]$ as the counter to denote the number of the indices of S-suffixes has been placed so far. Note that the RF-entry of this bucket is pointed by this counter (i.e. RF-pointer). Thus, we can place the index of the current S-suffix (S_i) into the RF-entry of this bucket in constant time, and then update the counter $\mathsf{SA}_S[m-2]$.

(2) If $\mathsf{SA}_S[m-1] = \mathsf{B_T}$ and $\mathsf{SA}_S[m - \mathsf{SA}_S[m-2] - 3] = \mathsf{B_H}$: In this case, we place the index of the third to last S-suffix (i.e. S_{m-2}) into the RF-entry of this bucket. Concretely, we shift the previous $m-3$ S-suffixes which stored in $\mathsf{SA}_S[1, \ldots, m-3]$ to $\mathsf{SA}_S[2, \ldots, m-2]$. Then, we place S_{m-2} into $\mathsf{SA}_S[1]$ and let $\mathsf{SA}_S[m-1] = \mathsf{R_2}$. This step takes $O(m)$ time since we shift $m-3$ S-suffixes.

(3) If $\mathsf{SA}_S[m-1] = \mathsf{R_2}$: In this case, we place the index of the second to last S-suffix (i.e. S_{m-1}) into the RF-entry of this bucket. We shift the previous $m-2$ S-suffixes which stored in $\mathsf{SA}_S[1, \ldots, m-2]$ to $\mathsf{SA}_S[2, \ldots, m-1]$. Then, we place S_{m-1} into $\mathsf{SA}_S[1]$ and let $\mathsf{SA}_S[0] = \mathsf{R_1}$. This step takes $O(m)$ time since we shift $m-2$ S-suffixes.

(4) Otherwise: In this case, we place the index of the last S-suffix (i.e. S_m) into the RF-entry of this bucket. First, we know the tail of the bucket indicated by our pointer data structure in constant time. Then, we search the entries before the tail one by one until that we find the special symbol $\mathsf{R_1}$. We let this entry to be S_m. This step takes $O(m)$ time since we search $m-1$ S-suffixes.

Note that this step uses $O(1)$ workspace since there is no bucket array and type array, and the space needed by our interior counter trick and pointer data structure is in SA_S. The purpose of the interior counter trick is to dynamic maintain the RF-pointers of the buckets. E.g., for a query of RF-entry for $\mathsf{suf}(j)$ in bucket $T[j]$, first we know the tail of the bucket $T[j]$ by the assumption, then

[5] We use at most five special symbols in this paper. The special symbol is only used to simplify the argument and we do not have to impose any additional assumption to accommodate these symbols (including the read-only general alphabets case). These special symbols can be handled using an extra $O(1)$ workspace. The details can be found in our full version [24].

we use the interior counter trick to indicate the RF-entry in this bucket. We have the following lemma.

Lemma 3. *If the tail of the bucket of any S-suffix is known, one can sort the S-suffixes from the sorted L-suffixes using the induced sorting step with the interior counter trick in linear time and $O(1)$ workspace.*

Note that in the induced sorting step, one uses the type array to identify whether the $\mathsf{suf}(j)$ is S-suffix or not. For removing the type array, we use the following Lemma 4 to identify the type of the L- or S-suffix in the induced-sorting step. Note that it is not hard to decide whether all S-suffixes in the current scanning bucket $T[\mathsf{SA}[i]]$ are already sorted or not by using an extra variable (reused for all buckets).

Lemma 4. *If $T[j] \neq T[\mathsf{SA}[i]]$, the type of $\mathsf{suf}(j)$ can be obtained immediately, where $j = \mathsf{SA}[i] - 1$. Otherwise $T[j] = T[\mathsf{SA}[i]]$ (this case $\mathsf{suf}(j)$ belongs to the current scanning bucket $T[\mathsf{SA}[i]]$), if all S-suffixes of T that belong to bucket $T[\mathsf{SA}[i]]$ are not already sorted, then the $\mathsf{suf}(j)$ is S-suffix.*

Get the Tails of the Buckets: Now, there is only one thing left: how to know the tails of the bucket of S-suffixes in the induced sorting step (this is the only assumption we used above). The purpose of the pointer data structure is to indicate the tails of the bucket of S-suffixes. However, the pointer data structure requires c_p words, where the value of c_p will be specified later. Thus, we need to divide this induced-sorting step into two stages. The first stage is to sort the first $n_S - c_p$ S-suffixes (i.e. the largest $n_S - c_p$ S-suffixes), where our pointer data structure exists. The second stage is to sort the last c_p S-suffixes, where there is no space for the pointer data structure.

The First Stage: Now, we construct our pointer data structure which supports to find the tails of the buckets in constant time. We store the pointer data structure in the tail of SA_S, recall that $\mathsf{SA}_S = \mathsf{SA}[n_L, \ldots, n-1]$. Now, we describe the details. We divide the S-suffixes of T into $4d$ parts according to their first characters, and construct the pointer data structure for each part respectively. The $4d$ parts are divided by $T[j] \in \left[\frac{i|\Sigma|}{4d} + 1, \frac{(i+1)|\Sigma|}{4d}\right]$, for $0 \leq i < 4d$. Let D_i denote the pointer data structure of the i-th part. We only show the details how we construct the pointer data structure D_0 as follows, since constructing D_i is similar for $0 < i < 4d$ (i.e., shift $T[j]$ with $\frac{i|\Sigma|}{4d}$).

(1) First, we let $\mathsf{SA}_S[i] = 1$ for all $i \in [1, \frac{|\Sigma|}{4d}]$. Then we scan T from right to left. For every S-type $T[i] \in [1, \frac{|\Sigma|}{4d}]$, we increase $\mathsf{SA}_S[T[i]]$ by one.
(2) Then we scan $\mathsf{SA}_S[1 \ldots \frac{|\Sigma|}{4d}]$ from left to right. We use a variable *sum* to count the sum, first initialize $sum = -1$. For each $\mathsf{SA}_S[i]$ which is being scanned, first let $sum = sum + \mathsf{SA}_S[i]$, then let $\mathsf{SA}_S[i] = sum$. Now, for any S-suffix $\mathsf{suf}(i)$ satisfying $T[i] \in [1, \frac{|\Sigma|}{4d}]$, $\mathsf{SA}_S[T[i]] - T[i]$ must indicate the tail of bucket $T[i]$ in SA_S. Since we want every entry in $\mathsf{SA}_S[1 \ldots \frac{|\Sigma|}{4d}]$ to be

distinct, we initialize $\mathsf{SA}_S[i] = 1$ for all $i \in [1, \frac{|\Sigma|}{4d}]$ in Step (1). Hence the tail of bucket $T[i]$ is $\mathsf{SA}_S[T[i]] - T[i]$.

(3) Finally, we construct D_0 for $\mathsf{SA}_S[1 \ldots \frac{|\Sigma|}{4d}]$ according to Lemma 5. D_0 uses at most $c(n + \frac{|\Sigma|}{4d})/\log n$ words space. We store D_0 in the tail of SA_S (i.e., $\mathsf{SA}_S[n_S - c(n + \frac{|\Sigma|}{4d})/\log n \ldots n_S - 1]$). D_0 supports to find the tail of the bucket of any S-suffix $\mathsf{suf}(i)$ satisfying $T[i] \in [1, \frac{|\Sigma|}{4d}]$ in constant time.

Lemma 5. *For any m distinct integers $0 \leq a_0 < a_1 \ldots < a_{m-1} \leq n$, where $m \leq n$ and $n > 1024$, one can construct a data structure using linear time (i.e., $O(n)$ time) and at most $cn/\log n$ words, where $1 < c < 2$, such that each query to the i-th smallest integer a_i ($\mathsf{select}(i)$) can be answered in constant time.*

The Lemma 5 is proved by using the classical select query in a bitmap (see e.g., [6,17,30]). The proof can be found in our full version [24]. After this step, the pointer data structure (i.e. D_i for all $0 \leq i < 4d$) is stored in $\mathsf{SA}_S[n_S - c_p \ldots n_S - 1]$, where $c_p = \lceil 4d \cdot (c(n + \frac{|\Sigma|}{4d})/\log n) \rceil \leq \lceil 5dcn/\log n \rceil$. Now, we have the following lemma.

Lemma 6. *We can construct the pointer data structure in linear time, and this pointer data structure uses at most c_p words and can support to find the bucket tail of any S-suffix in constant time.*

Now according to Lemmas 3, 4 and 6, we can sort the first largest $n_S - c_p$ S-suffixes from the sorted L-suffixes which stored in SA_L using the induced sorting step.

The Second Stage: Now, we describe the details to sort the last c_p S-suffixes which is occupied by our pointer data structure. First, we move the sorted largest $n_S - c_p$ S-suffixes to the tail of SA_S, i.e., $\mathsf{SA}_S[c_p, n_S - 1]$. Then we scan the T from right to left to place the smallest c_p S-suffixes into $\mathsf{SA}_S[0, c_p - 1]$. Now, we use merge sort with the in-place linear time merging algorithm [37] to sort these c_p S-suffixes, the sorting key for each S-suffix is its beginning character. After this sorting step, these c_p S-suffixes have been placed in their corresponding buckets in $\mathsf{SA}_S[0, c_p - 1]$. Note that we can use the same sorting step (which we used for sorting the first $n_S - c_p$ S-suffixes) to sort the last c_p S-suffixes without the pointer data structure.

The key point is that we can use the binary search (instead of the pointer data structure) to find the tails of the bucket for these c_p S-suffixes, since $c_p \leq \lceil 5dcn/\log n \rceil$ is small enough (i.e. $c_p \log n = O(n)$) to maintain that the time complexity of our algorithm is $O(n)$. Using the binary search to extend interior counter trick is not very difficult, one can find the details in our full version [24] where we induced sort all L-suffixes from the sorted S-suffixes for the read-only general alphabets.

After the second stage, all n_S S-suffixes are sorted in SA_S. Now we have all sorted L-suffixes in SA_L (i.e., $\mathsf{SA}[0 \ldots n_L - 1]$) and all sorted S-suffixes in SA_S (i.e., $\mathsf{SA}[n_L \ldots n - 1]$). Then, we use the stable, in-place, linear time merging algorithm [37] to merge the ordered SA_L and SA_S (the merging key for $\mathsf{SA}[i]$ is

$T[\mathsf{SA}[i]]$, i.e., the first character of $\mathsf{suf}(\mathsf{SA}[i])$). After this merging step, all suffixes of T have be sorted in $\mathsf{SA}[0 \ldots n-1]$.

Theorem 3 (Main Theorem). *Our Algorithm takes $O(n)$ time and $O(1)$ workspace to compute the suffix array of string T over integer alphabets Σ, where T is read-only and $|\Sigma| = O(n)$.*

4 Conclusion

In this paper, we present the optimal in-place algorithms for suffix sorting over (read-only) integer alphabets and read-only general alphabets. All of them are optimal both in time and space. Concretely, we provide the first optimal linear time in-place suffix sorting algorithm for (read-only) integer alphabets. Our algorithms solve the open problem posed by Franceschini and Muthukrishnan in ICALP 2007 [11]. For the read-only general alphabets (the details of this part can be found in our full version [24]), we provide simple sorting steps to obtain an optimal in-place $O(n \log n)$ time suffix sorting algorithm, which recovers the result obtained by Franceschini and Muthukrishnan [11] which was an open problem posed by Manzini and Ferragina [28].

Acknowledgments. This research is supported in part by the National Basic Research Program of China Grant 2015CB358700, the National Natural Science Foundation of China Grant 61772297, 61632016, 61761146003, and a grant from Microsoft Research Asia. The authors would like to thank Ge Nong for his help in our experiments, and Gonzalo Navarro for helpful suggestions.

References

1. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: The enhanced suffix array and its applications to genome analysis. In: Guigó, R., Gusfield, D. (eds.) WABI 2002. LNCS, vol. 2452, pp. 449–463. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45784-4_35
2. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. J. Discret. Algorithms **2**(1), 53–86 (2004)
3. Baron, D., Bresler, Y.: Antisequential suffix sorting for bwt-based data compression. IEEE Trans. Comput. **54**(4), 385–397 (2005)
4. Burkhardt, S., Kärkkäinen, J.: Fast lightweight suffix array construction and checking. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 55–69. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44888-8_5
5. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Technical report 124 (1994)
6. Clark, D.: Compact pat trees. Ph.D. thesis, University of Waterloo (1996)
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press, Cambridge (2001)
8. Dhaliwal, J., Puglisi, S.J., Turpin, A.: Trends in suffix sorting: a survey of low memory algorithms. In: Proceedings of the Thirty-fifth Australasian Computer Science Conference, vol. 122, pp. 91–98. Australian Computer Society, Inc. (2012)

9. Farach, M.: Optimal suffix tree construction with large alphabets. In: Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS), pp. 137–143. IEEE (1997)
10. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS), pp. 390–398. IEEE (2000)
11. Franceschini, G., Muthukrishnan, S.: In-place suffix sorting. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 533–545. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73420-8_47
12. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM J. Comput. **35**(2), 378–407 (2005)
13. Hon, W.K., Sadakane, K., Sung, W.K.: Breaking a time-and-space barrier in constructing full-text indices. In: Proceedings of the 44th Annual Symposium on Foundations of Computer Science (FOCS), pp. 251–260. IEEE (2003)
14. Huo, H., Chen, L., Vitter, J.S., Nekrich, Y.: A practical implementation of compressed suffix arrays with applications to self-indexing. In: Data Compression Conference (DCC), pp. 292–301. IEEE (2014)
15. Huo, H., et al.: CS2A: a compressed suffix array-based method for short read alignment. In: Data Compression Conference (DCC), pp. 271–278. IEEE (2016)
16. Itoh, H., Tanaka, H.: An efficient method for in memory construction of suffix arrays. In: String Processing and Information Retrieval Symposium, 1999 and International Workshop on Groupware, pp. 81–88. IEEE (1999)
17. Jacobson, G.: Space-efficient static trees and graphs. In: Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS), pp. 549–554. IEEE (1989)
18. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 943–955. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-45061-0_73
19. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM (JACM) **53**(6), 918–936 (2006)
20. Kim, D.K., Jo, J., Park, H.: A fast algorithm for constructing suffix arrays for fixed-size alphabets. In: Ribeiro, C.C., Martins, S.L. (eds.) WEA 2004. LNCS, vol. 3059, pp. 301–314. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24838-5_23
21. Kim, D.K., Sim, J.S., Park, H., Park, K.: Linear-time construction of suffix arrays. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 186–199. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44888-8_14
22. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 200–210. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44888-8_15
23. Larsson, N.J., Sadakane, K.: Faster suffix sorting. Theor. Comput. Sci. **387**(3), 258–272 (2007)
24. Li, Z., Li, J., Huo, H.: Optimal in-place suffix sorting. arXiv preprint arXiv:1610.08305 (2016)
25. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. In: Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 319–327. Society for Industrial and Applied Mathematics (1990)

26. Maniscalco, M.A., Puglisi, S.J.: Faster lightweight suffix array construction. In: Proceedings of International Workshop On Combinatorial Algorithms (IWOCA), pp. 16–29. Citeseer (2006)
27. Maniscalco, M.A., Puglisi, S.J.: An efficient, versatile approach to suffix sorting. J. Exp. Algorithmics (JEA) **12**, 1–2 (2008)
28. Manzini, G., Ferragina, P.: Engineering a lightweight suffix array construction algorithm. In: Möhring, R., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 698–710. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45749-6_61
29. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. ACM (JACM) **23**(2), 262–272 (1976)
30. Navarro, G., Providel, E.: Fast, small, simple rank/select on bitmaps. In: Klasing, R. (ed.) SEA 2012. LNCS, vol. 7276, pp. 295–306. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30850-5_26
31. Nong, G.: Practical linear-time O (1)-workspace suffix sorting for constant alphabets. ACM Trans. Inf. Syst. (TOIS) **31**(3), 15 (2013)
32. Nong, G., Zhang, S.: Optimal lightweight construction of suffix arrays for constant alphabets. In: Dehne, F., Sack, J.-R., Zeh, N. (eds.) WADS 2007. LNCS, vol. 4619, pp. 613–624. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73951-7_53
33. Nong, G., Zhang, S., Chan, W.H.: Linear suffix array construction by almost pure induced-sorting. In: Data Compression Conference (DCC), pp. 193–202. IEEE (2009)
34. Nong, G., Zhang, S., Chan, W.H.: Two efficient algorithms for linear time suffix array construction. IEEE Trans. Comput. **60**(10), 1471–1484 (2011)
35. Puglisi, S.J., Smyth, W.F., Turpin, A.H.: A taxonomy of suffix array construction algorithms. ACM Comput. Surv. (CSUR) **39**(2), 4 (2007)
36. Sadakane, K.: A fast algorithm for making suffix arrays and for burrows-wheeler transformation. In: Data Compression Conference (DCC), pp. 129–138. IEEE (1998)
37. Salowe, J., Steiger, W.: Simplified stable merging tasks. J. Algorithms **8**(4), 557–571 (1987)
38. Schürmann, K.B., Stoye, J.: An incomplex algorithm for fast suffix array construction. Softw.: Pract. Exp. **37**(3), 309–329 (2007)
39. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Trans. Inf. Theory **24**(5), 530–536 (1978)