In-place Linear Time Suffix Array Construction

Keisuke Goto[1]

1 **Fujitsu Laboratories Ltd,.**
`goto.keisuke@jp.fujitsu.com`

Abstract

We present a new linear time algorithm for computing the suffix array of a given string of length N on an alphabet of size σ using $O(\log N)$ bits of extra working space. Our algorithm improves the space requirement of the previous most space efficient algorithm [Nong, TOIS 2013] using $\sigma \log N + O(\log N)$ bits of extra working space.

リファレンス以外で 10 ページまで

Keywords and phrases Suffix Array, Pattern Matching

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 TODO

1. correct position とは
2. in-place とは

2 Introduction

The suffix array is an index that stores all suffixes of a given string of length N in lexicographic order. It is an important tool for string processing since many applications such as pattern matching, data mining, and many of pattern discoveries, can be efficiently solved by using suffix arrays.

In this paper, we propose first *in-place* suffix array construction algorithm. Here in-place means that we only require $O(\log N)$ bits except the space of the suffix array and given string of length N. While suffix tree constitute of tree structure and require huge amount of space, suffix array consititute of integer array of length N and require less space. However, most algorithms needs additional working space more than the suffix array, and it becomes an obstacle to treat enormous sized data.

Recently Nong propose a *near in-place* algorithm [19] based on induced sorting which can compute the suffix array in linear time and use $\sigma \log N + O(\log N)$ bits of working space, where σ is the number of characters which appear in the given string. This accomplished significant improvements since it behaves as in-place if σ is small. However, when σ getting close to N, the space requirement is still close to $N \log N$ bits.

If we have any in-place sorting algorithm which runs in $O(N \log N)$ time, we can obtain in-place suffix array construction algorithm which runs in $O(N^2 \log N)$ time since any ordering of two suffixes can be obtained in $O(N)$ time. However, its the large time complexity does not suit of our issue for enormous sized data.

Is it possible to construct suffix array in-place? We answer this question, yes. We propose the first *truly* in-place suffix array construction algorithm which is based on Nong's algorithm [19] using induced sorting technique. His algorithm ditermines the rank for each suffixes from sorted subset of suffixes, and store them to correct positions in the suffix array. In the process, his algorithm requires additional integer array **bkt** of length σ to store next inserting position of suffixes starting with each character. Therefore his algorithm requires $\sigma \log N$ bits of extra working space.

We carefully manage **bkt** in the somewhere which have not inserted suffixes in the suffix array, and construct suffix arrays. To do this we firstly construct suffix arrays for subset of suffixes, and then merge them by using in-place merge algorithms [3].

In the rest of paper, we introduce suffix array construction algorithm which uses $\sigma \log N + N + O(\log N)$ bits which is based on induced sorting in Section 5, and how to reduce the space to $O(\log N)$ bits of working space.

3 Introduction2

Most suffix array construction algorithms use the extra working space except the given string and the suffix array to be computed. It can be crucial problem to treat a string of large length. The big goal is to propose to compute the suffix array *in-place*, where in-place means to require only constant word sized working space $O(\log N)$ bits.

The algorithms [**?**] based on induced copying accomplish this, but it takes $O(N^2 \log N)$ time in the worst case. Recently Nong propose a *near in-place* algorithm [19] based on induced sorting which can compute the suffix array in linear time and use $\sigma \log N + O(\log N)$ bits of working space, where σ is the number of characters which appear in the given string. However, when σ getting close to N, the algorithm requires the almost same space of the suffix array as working space.

We propose the first *truly* in-place linear time algorithm to compute the suffix array which is the extension of Nong's algorithm. Nong's algorithm is based on the following induced sorting approach:

1. Select at most half the number of suffixes from all suffixes, and then recursively sorts them.
2. Induce their preceding suffixes from the suffixes sorted in the previous step, and put them to the correct position in the suffix array, where induce means to obtain the rank of the suffix from another.

Nong uses the extra data structure, whose size is $\sigma \log N$ bits, to indicate the next inserting position in the suffix array for the suffixes preceding each character. The algorithm use the data structure implicitly and runs in-place in the recursive step, but it does not the first step. We propose a new technique to use the data structure implicitly in the first step so that the algorithm runs in-place in the whole step.

We assume word-RAM model of word size $w = O(\log N)$ bits.

Moreover, the stack space for the recursive is not counted

A consumption is its required working space to compute the suffix array. So far, many researchers tackle the problem. We propose the first *in-place* linear time suffix array construction algorithm whose space requirement is better than any previous ones. Here, in-place means that the algorithm requires only $O(\log N)$ bits of working space, where N is the length of the input string.

—

The suffix array in a fundamental data structured to be applied to many applications such as pattern matching, data mining, and many of pattern discoveries. A consumption is its required working space to compute the suffix array. So far, many researchers tackle the problem. We propose the first *in-place* linear time suffix array construction algorithm whose space requirement is better than any previous ones. Here, in-place means that the algorithm requires only $O(\log N)$ bits of working space, where N is the length of the input string.

Since firstly proposed the linear time suffix array construction algorithm, many kinds of algorithm have been proposed so far [**?**, **?**, **?**, **?**, **?**, 19].

4 Preliminaries

Let Σ be a finite alphabet, and let $\sigma = |\Sigma|$. An element of Σ^* is called a string. The length of a string $\mathbf{T}$ is denoted by $|\mathbf{T}|$. The empty string ϵ is the string of length 0, namely $|\epsilon| = 0$. For a string $\mathbf{T} = XYZ$, X, Y and Z are called a *prefix*, *substring*, and *suffix* of $\mathbf{T}$, respectively.

The i-th character of a string $\mathbf{T}$ is denoted by $\mathbf{T}[i]$ for $1 \leq i \leq |\mathbf{T}|$, and the substring of $\mathbf{T}$ that begins at position i and ends at position j is denoted by $\mathbf{T}[i..j]$ for $1 \leq i \leq j \leq |\mathbf{T}|$. For convenience, we assume that $\mathbf{T}[|\mathbf{T}|] = \$$, where $\$$ is a special character that does not occur elsewhere in the string $\mathbf{T}[1..|\mathbf{T}| - 1]$.

For a string $\mathbf{T}$ of length N, a suffix starting at i is denoted $\mathbf{T}_i$. For $1 \leq i \leq N$, $\mathbf{T}_i$ is called Small type suffix (S-suffix) if $i = N$ or $\mathbf{T}_i$ is lexicographically smaller than $\mathbf{T}_{i+1}$, and Large type suffix (L-suffix) otherwise. S-suffix $\mathbf{T}_i$ is also called a Left-Most-S-suffix(LMS-suffix) if $i = 1$ or $\mathbf{T}_{i-1}$ is L-suffix, and similarly L-suffix $\mathbf{T}_i$ is also called a Left-Most-L-suffix(LML-suffix) if $i = 1$ or $\mathbf{T}_{i-1}$ is S-suffix. We denote $suf(all)$ is the set of all suffixes which appears in $\mathbf{T}$, and also denote $suf(L)$, $suf(S)$, $suf(LML)$, and $suf(LMS)$ are the set of all L, S, LML, LMS suffixes which appear in $\mathbf{T}$, respectively. Note that the size of either $suf(L)$ or $suf(S)$ must be less than or equal to $N/2$ since all of suffixes belong to either one. Moreover the size of $suf(LML)$ and $suf(LMS)$ also must be less than or equal to the size of the smallest of $suf(L)$ and $suf(S)$.

The suffix array $\mathbf{SA}_M$ of M which is a subset of $suf(all)$ is an array of length $|M|$ such that for any $1 \leq i \leq |M|$, $\mathbf{SA}[i] = j$ indicates that $\mathbf{T}_j \in M$ is the i-th lexicographically smallest suffix of M. When the context is clear, we denote $\mathbf{SA}$ as $\mathbf{SA}_{suf(all)}$. For each character c, the maximum interval in which the heading characters of suffixes are equal to c is called c-interval. Since S, L-suffixes are larger and smaller than succeeding one, respectively, for a character c, L-suffixes starting with c always located before than S-suffixes starting with c.

KG Note: 両方を述べる時 S-suffix と L-suffix のどっちを先行させるか統一したい

5 Induced Sorting

In this section, we explain an induced sorting algorithm to compute $\mathbf{SA}$ for a string $\mathbf{T}$ of length N. Most induced sorting algorithms explicitly or implicitly need two integer arrays $\mathbf{X}$ and $\mathbf{bkt}$, and one boolean array $\mathbf{type}$, where $\mathbf{X}$ is an integer array of length N to be $\mathbf{SA}$ when algorithm ends, $\mathbf{bkt}$ is an integer array of length σ such that $\mathbf{bkt}[c]$ indicate the next inserting position of a suffix starting with c in $\mathbf{SA}$ [1], $\mathbf{type}$ is an boolean array of length N indicating their types such that $\mathbf{type}[i]$ is L if $\mathbf{T}_i$ is L-suffix and S otherwise. In the initial state of each step, for all $c \in \Sigma$, $\mathbf{bkt}[c]$ indicate the left most or right most position of c-interval in $\mathbf{SA}$, and it can be computed in $O(N)$ time by counting all characters in $\mathbf{T}$ and accumulating the values in lexicographic order.

The overview of the induced sorting algorithm is below:

[1] Nong's algorithm explicitly uses $\mathbf{SA}$ and $\mathbf{bkt}$ and implicitly use $\mathbf{type}$, and needs $N \log N + \sigma \log N + O(\log N)$ bits space in total

1. Sort the LMS-suffixes.
   For two LMS-suffixes $\mathbf{T}_i$ and $\mathbf{T}_j$, we call $\mathbf{T}[i..j]]$ is LMS-substring if there is any LMS-suffix between them. We sort all LMS-substrings, and check whether they are unique or not. If so, LMS-suffixes are also sorted. Otherwise, we make a new string that corresponds to all LMS-substrings in $\mathbf{T}$ are replaced to those ranks among them, and store the right hand side of $\mathbf{X}$. Note that the length of new string is at most $N/2$. We recursively make its suffix array and store the left hand side of $\mathbf{X}$.
2. Arrange the LMS-suffixes
   We assume that $\mathbf{X}[1..|C(suf(LMS))|] = \mathbf{SA}_{suf(LMS)}$, $\mathbf{X}[|\mathbf{SA}_{suf(LMS)}|+1..N] =$ EMPTY, and $\mathbf{bkt}[c]$ indicate the right most position of c-interval for all $c \in \Sigma$. By a right to left scan on $\mathbf{X}[1..C(suf(LMS))]$, for a LMS-suffix $\mathbf{T}_{\mathbf{X}[i]}$ starting with c, we put it to $\mathbf{X}[\mathbf{bkt}[c]]$ which is the right most empty position of c-interval, update $\mathbf{X}[i] =$ EMPTY. We then update $\mathbf{bkt}[c] = \mathbf{bkt}[c] - 1$ to indicate the right most empty position of c-interval.
3. Sort the L-suffixes
   We assume that $\mathbf{bkt}[c]$ indicate the left most position of c-interval for all $c \in \Sigma$. By a left to right scan on $\mathbf{X}$, if $\mathbf{X}[i]$ is non-empty and $\mathbf{T}_{\mathbf{X}[i]} \neq \mathbf{T}_1$, we determine the type of $\mathbf{T}_{\mathbf{X}[i]-1}$ by $\mathbf{type}[\mathbf{X}[i]-1]$. If $\mathbf{T}_{\mathbf{X}[i]-1}$ is L-suffix starting with c, we put it to $\mathbf{X}[\mathbf{bkt}[c]]$ which is the left most empty position of the c-interval. We then update $\mathbf{bkt}[c] = \mathbf{bkt}[c] + 1$ to indicate the left most empty position of c-interval.
4. Sort the S-suffixes
   It can be done in similar way of the sorting the L-suffixes.

▶ Correctness 1. In Step 3, we access all $\mathbf{T}_i \in suf(LMS) \cup suf(L)$ in lexicographic order and put a L-suffix $\mathbf{T}_{i-1}$ starting with c to the left most empty position of c-interval in $\mathbf{X}$. For any L-suffix $\mathbf{T}_{j-1}$ starting with c which is inserted before $\mathbf{T}_{i-1}$, it must be smaller than $\mathbf{T}_{i-1}$ since $\mathbf{T}_j$ must be accessed before and lexicographically smaller than $\mathbf{T}_i$. Therefore all induced L-suffixes are stored and sorted in $\mathbf{X}$.

In each step, the algorithm scan the whole of $\mathbf{X}$, $\mathbf{bkt}$, and $\mathbf{T}$ in constant times, so it takes $O(N)$ time in total.

6 In-Place Algorithm when $2\sigma \leq N$

We assume $2\sigma \leq N$.

We propose a new induced sorting algorithm without **bkt** and **type** explicitly. In contrast to the algorithm in Section 5 directory puts induced suffixes to the correct positions in $\mathbf{SA}$, our algorithm constructs some suffix arrays for divided subsets of suffixes in-place, and then merge them by using the stable merge algorithm [?] for two sorted arrays which runs linear time and in-place.

▶ **Theorem 1** ([?]). *For two integer arrays $\mathbf{Y}_1[1..N_1]$ and $\mathbf{Y}_2[1..N_2]$ whose values are sorted in each of them and which are stored in $\mathbf{Y}$ as $\mathbf{Y}[1..N_1] = \mathbf{Y}_1$ and $\mathbf{Y}[N_1+1..(N_1+N_2)] = \mathbf{Y}_2$. There is an algorithm to merge $\mathbf{Y}_1$ and $\mathbf{Y}_2$, and sort the values in $\mathbf{Y}$ stably in $O(N_1+N_2)$ time and in-place.*

This merge strategy gives following two benefits to us:

1. We can easily determine the type of suffixes.
   In each step of our algorithm, we have at most three suffix arrays $\mathbf{SA}_{suf(L')}$, $\mathbf{SA}_{suf(S')}$, and $\mathbf{SA}_{others}$. $\mathbf{SA}_{suf(L')}$ and $\mathbf{SA}_{suf(S')}$ contain only L-suffixes and S-suffixes, respectively, so we can obtain the types of them when accessing them. $\mathbf{SA}_{others}$ contains only

the largest L-suffixes or the smallest S-suffixes starting with each character c, so we can easily determine the type of them by comparing heading characters between them and the their preceding suffixes starting with another character. 集合表記を使わないほうが良いのかも。

2. We can store **bkt** in the empty space of $\mathbf{X}$.
   Our algorithm stores induced L, S-suffixes into continuous space in $\mathbf{X}$ as $\mathbf{SA}_{suf(L')}$ and $\mathbf{SA}_{suf(S')}$. In other words, empty space is also continuous, and we use the space to store **bkt**. However, this may cause overflow since the total size of them may exceed N. We can avoid the problem by detecting the values of **bkt** which will never be used in future, and reusing the space to store suffixes.

Since sorting LMS-suffixes can be computed by the combination of the techniques of sorting L, S-suffixes, we firstly explain sorting L, S-suffixes and later LMS-suffixes.

6.1 Sort the L-suffixes

We assume that $\mathbf{X}[1..C(suf(LMS))] = \mathbf{SA}_{suf(LMS)}$. We use $\mathbf{X}[N - \sigma + 1..N]$ as **bkt** and compute $\mathbf{SA}_L$ from $\mathbf{SA}_{suf(LMS)}$ and **bkt**. The problem is that the empty space of $\mathbf{X}$ may not be sufficient to store $\mathbf{SA}_L$. To avoid the problem, our algorithm uses $\mathbf{SA}_{suf(LMSx)}$ rather than $\mathbf{SA}_{suf(LMS)}$, and computes $\mathbf{SA}_{suf(Lx)}$ rather than $\mathbf{SA}_{suf(L)}$, where $suf(LMSx)$ and $suf(Lx)$ are a subset of $suf(LMS)$ and $suf(L)$, respectively. In the process, we carefully mange **bkt** so that it indicates not only the pointer to a next inserting position of L-suffix but also LMS, L-suffixes which are not belong to $suf(LMSx)$ nor $suf(Lx)$, and the total space does not exceed the space of $\mathbf{X}$.

The progress of $\mathbf{X}$ in the algorithm to compute $\mathbf{SA}_L$ is shown in Figure 1. The algorithm uses only following data and not any more in the process.

- $suf(Lx')$ is the set of all suffixes that are the largest ones starting with c for each character c.
- $suf(Lx) = suf(L) - suf(Lx')$.
- $suf(LMSx')$ is the set of all suffixes that are the smallest ones starting with c and there are not any L-suffixes starting with c for each character c.
- $suf(LMSx) = suf(LMS) - suf(LMSx')$.
- $\mathbf{bkt}[c]$ indicates the left most empty position of c-interval in $\mathbf{SA}_{suf(Lx)}$ if L-suffix starting with c exists, and EMPTY otherwise.
- $\mathbf{bkt}_{lms}[c]$ indicates $\mathbf{bkt}[c]$ if a L-suffix starting with c exists, and the largest LMS-suffix starting with c otherwise.
- $\mathbf{bkt}_{lx}[c]$ indicates the largest L-suffix starting with c if such L-suffix exists, and the smallest LMS-suffix starting with c.

▶ **Lemma 2.** *In the process in Figure 1, workspace in* $\mathbf{X}$ *does not exceed* N.

Proof. Since $2|suf(LMS)| \leq N$, and $2\sigma \leq N$, the sum of sizes $\mathbf{SA}_{suf(LMS)}$ and **bkt** does not exceed N. Since $|suf(Lx')| + |suf(LMSx')| = \sigma$, the sum of sizes $\mathbf{SA}_{suf(LMSx)}$, $\mathbf{SA}_{suf(Lx)}$, and $\mathbf{bkt}_{lx}$ does not exceed N. Trivially, the total sizes in other steps do not exceed N. ◀

The rest of question is how to progress between the steps in Figure 1 in-place. To compute $\mathbf{SA}_L$, the algorithm treats $\mathbf{X}[N - \sigma + 1..N]$ as **bkt**, $\mathbf{bkt}_{lms}$, and $\mathbf{bkt}_{lx}$, and they are transformed to others gradually. To avoid the confusion, we will denote all of them as **bkt** in the following explanations. 個々の部分どうする？

1. Compute $\mathbf{bkt}$.
   $\mathbf{bkt}$ can be computed in almost similar way of Section 5. The difference is to count only characters that appear as the heading of L-suffixes in $\mathbf{T}$. When accumulating values, if the $\mathbf{bkt}[c] = 0$, it indicates that there are not any L-suffixes starting with c, so set $\mathbf{bkt}[c] = \text{EMPTY}$. Otherwise, let $count_c$ be the number of L-suffixes starting with c which is stored in $\mathbf{bkt}[c]$, set accumulated value to $\mathbf{bkt}[c]$ as the beginning position of c-interval in $\mathbf{SA}_{suf(Lx)}$, and accumulate $count_c - 1$ not $count_c$ to to indicate the interval of $\mathbf{SA}_{suf(Lx)}$ rather than $\mathbf{SA}_{suf(L)}$.
2. Compute $\mathbf{SA}_{suf(LMSx)}$.
   By a right to left scan of $\mathbf{SA}_{suf(LMS)}$ and $\mathbf{bkt}$, if $\mathbf{bkt}[c] = \text{EMPTY}$, we swap $\mathbf{bkt}[c]$ and the left most (smallest) LMS-suffix starting with c. After that, $\mathbf{bkt}$ becomes $\mathbf{bkt}_{lms}$, and $\mathbf{SA}_{suf(LMS)}$ stores $suf(LMSx)$ sparsely. By a left to right scan on $\mathbf{SA}_{suf(LMS)}$, we left justify non-empty elments on $\mathbf{SA}_{suf(LMS)}$ and make $\mathbf{X}[1..|suf(LMSx)|] = \mathbf{SA}_{suf(LMSx)}$.
3. Compute $\mathbf{SA}_{suf(Lx)}$ and $\mathbf{bkt}_{lx}$.
   a. Access all LMS, L-suffixes $\mathbf{T}_i$ lexicographically.
      Since $\mathbf{bkt}_{lx}$ contains the smallest LMS-suffix or the largest L-suffix starting with c for each character c, we can access LMS, L-suffixes lexicographically by accessing $\mathbf{SA}_{suf(Lx)}$, $\mathbf{bkt}_{lx}$, and $\mathbf{SA}_{suf(LMSx)}$ in this order for each increasing character.
   b. Determine whether $\mathbf{T}_{i-1}$ is L-suffix or not.
      When we encounter a suffix $\mathbf{T}_i$ stored in either $\mathbf{SA}_{suf(Lx)}$, $\mathbf{bkt}_{lx}$, or $\mathbf{SA}_{suf(LMSx)}$, if $\mathbf{T}_i$ is from $\mathbf{SA}_{suf(LMSx)}$ and $\mathbf{SA}_{suf(Lx)}$, we know its type, so easily determine the type of $\mathbf{T}_{i-1}$ by comparing the heading characters. Otherwise, heading characters of $\mathbf{T}_i$ and $\mathbf{T}_{i-1}$ must be different since $\mathbf{T}_i$ is the largest L-suffix or the smallest S-suffix for a character. We can determine the type of $\mathbf{T}_{i-1}$ by comparing the heading characters.
   c. Put L-suffix $\mathbf{T}_{i-1}$ starting with c into $\mathbf{SA}_{suf(Lx)}$ or $\mathbf{bkt}_{lx}$.
      We try to put it to $\mathbf{SA}_{suf(Lx)}[\mathbf{bkt}_{lms}[c]]$. If $\mathbf{SA}_{suf(Lx)}[\mathbf{bkt}_{lms}[c]]$ is empty, we simply put $\mathbf{T}_{i-1}$ there. Since there is not sufficient space to store all L-suffixes starting with c by one, and conflicts may occur when trying to put $\mathbf{T}_{i-1}$ to $\mathbf{SA}_{suf(Lx)}[\mathbf{bkt}_{lms}[c]]$. If conflicts occur, let $\mathbf{T}_j = \mathbf{T}_{\mathbf{SA}_{suf(Lx)}[\mathbf{bkt}_{lms}[c]]}$. If $\mathbf{T}_j$ is starting with c, $\mathbf{T}_{i-1}$ is the largest L-suffix starting with c, then put $\mathbf{T}_{i-1}$ to $\mathbf{bkt}_{lx}[c]$. If $\mathbf{T}_j$ is not starting with c, it must be the largest L-suffix for a preceding character of c [2] or the smallest L-suffix for a succeeding character of c. In either case, we put the largest one to $\mathbf{SA}_{suf(Lx)}[\mathbf{bkt}_{lms}[c]]$, and the other starting with c' to $\mathbf{bkt}_{lx}[c']$.

   Though $\mathbf{bkt}_{lms}$ is actually same array of $\mathbf{bkt}_{lx}$ located at $\mathbf{X}[N - \sigma + 1..N]$, those values are not mistaken each other in the above description. If there are not any L-suffixes starting with c, $\mathbf{bkt}_{lms}[c]$ is identical to $\mathbf{bkt}_{lx}[c]$, and mistake does not occur for such c. Consider the other case. In Step 3a, when accessing $\mathbf{bkt}_{lx}[c]$, all LMS, L-suffixes which are smaller than the largest L-suffix starting with c are accessed and induced those preceding suffixes. Therefore $\mathbf{bkt}_{lx}[c]$ must store the largest L-suffix starting with c if such suffix exists, and the smallest LMS-suffix for c otherwise, and mistake does not occur. In Step 3c, we put a L-suffix starting with c to $\mathbf{bkt}_{lx}[c]$ only if it is the largest L-suffix for c, namely last induced L-suffix for c. Therefore, $\mathbf{bkt}_{lms}[c]$ will never be used in future, and mistake does not occur.

[2] In this case, L-suffixes starting with c are only $\mathbf{T}_i$

4. Compute $\mathbf{SA}_{suf(Lx')}$.
   By a right to left scan on $\mathbf{T}$, we access all LMS-suffix $\mathbf{T}_i$ starting with c, and check whether it is stored in $\mathbf{bkt}_{lx}[c]$. If so, we set $\mathbf{bkt}_{lx}[c] = \text{EMPTY}$. After that, $\mathbf{bkt}_{lx}$ contains only $suf(Lx')$ sparsely. By a left to right scan on $\mathbf{bkt}_{lx}$, the algorithm left justify non-empty elements, and make $\mathbf{SA}_{suf(Lx')}$.
5. Compute $\mathbf{SA}_L$.
   Important observation is that for a suffix $\mathbf{T}_i$ in $\mathbf{SA}_{suf(Lx)}$ and $\mathbf{T}_j$ in $\mathbf{SA}_{suf(Lx')}$ both of which starting with the same character, $\mathbf{T}_i < \mathbf{T}_j$ is always hold since $\mathbf{SA}_{suf(Lx')}$ contains the largest L-suffix for the character. Moreover, $\mathbf{SA}_{suf(Lx)}$ and $\mathbf{SA}_{suf(Lx')}$ is arranged in this order in $\mathbf{X}$. We can obtain $\mathbf{SA}_{suf(L)}$ by stable merging of $\mathbf{SA}_{suf(Lx)}$ and $\mathbf{SA}_{suf(Lx')}$ concentrating only their heading characters.

From Theorem 1, we can merge $\mathbf{SA}_{lx}$ and $\mathbf{SA}_{suf(LMSx)}$, and compute $\mathbf{SA}_l$ in $O(N)$ time and $O(\log N)$ bits of extra working space.

6.2 Sort the S-suffixes

The procedure to sort S-suffixes is almost same of one to sort L-suffixes in Section 6.1. The progress of $\mathbf{X}$ is shown in Figure 2.

Let $suf(Lz')$ be a set of all L-suffixes which is the largest L-suffix starting with c that any S-suffix starting with c does not appear in $\mathbf{T}$, $suf(Lz)$ be $suf(L) - suf(Lz')$, $suf(Sz')$ be a set of all S-suffixes which is the smallest S-suffixes starting with c for each character c, and $suf(Sz)$ be $suf(S) - suf(Sz')$. We compute $\mathbf{SA}_{suf(Sz)}$ rather than $\mathbf{SA}_{suf(S)}$ from $\mathbf{SA}_{suf(Lz)}$ rather than $\mathbf{SA}_{suf(L)}$. It can be computed in similar way of Section 6.1. The difference $\mathbf{bkt}$ is located just right to $\mathbf{SA}_{setzL}$ and store $\mathbf{SA}_{suf(Sz)}$ to $\mathbf{X}[N - |suf(Sz)| + 1..N]$. After computing $\mathbf{bkt}$ stores $suf(Lz')$ and $suf(Sz')$ in lexicographic order as like $\mathbf{bkt}_{lx}$ stores $suf(Lx')$ and $suf(Sx')$ in lexicographic order. Now $\mathbf{SA}_{suf(Lz)}$, $\mathbf{SA}_{suf(Lz') \cup suf(Sz')}$, and $\mathbf{SA}_{suf(Sz)}$ is located in this order in $\mathbf{X}$. Since any suffix $\mathbf{T}_i$ in $\mathbf{SA}_{suf(Lz)}$, any suffix $\mathbf{T}_j$ in $\mathbf{SA}_{suf(Lz)}$, and any suffix $\mathbf{T}_k$ in $\mathbf{SA}_{suf(Lz)}$ all of them starting with same character c, it holds $\mathbf{T}_i < \mathbf{T}_j < \mathbf{T}_k$. Therefore we can merge them and make $\mathbf{SA}$.

6.3 $C(suf(L)) > C(suf(S))$

6.4 $2\sigma > N$

If we allow to modify given string $\mathbf{T}$, we can compute $\mathbf{SA}$ of $\mathbf{T}$ by Nong's algorithm [19] for modified string $\mathbf{T}'$. For a character $c = \mathbf{T}[i]$, $\mathbf{T}'[c] = |\{\mathbf{T}_j | 1 \le j \le N, \mathbf{T}_j < \mathbf{T}_i\}|$ if $\mathbf{T}_i$ is L-suffix, and $\mathbf{T}'[c] = |\{\mathbf{T}_j | 1 \le j \le N, i \ne j, \mathbf{T}_j \le \mathbf{T}_i\}|$ otherwise. They are correspond to the beginning and ending positions of c-interval. $\mathbf{T}'$ has a property which is part of $\mathbf{bkt}$. Nong's algorithm implicitly use $\mathbf{bkt}$ by using this property, and compute $\mathbf{SA}$ of $\mathbf{T}'$. Since all L-suffixes starting with c is less than all S-suffixes starting with c, $\mathbf{SA}$ of $\mathbf{T}'$ is identical to $\mathbf{SA}$ of $\mathbf{T}$. T' can be computed by counting all characters in similar way of [19].

Restoring $\mathbf{T}$ from $\mathbf{T}'$ is not trivial since we modify same character $c = \mathbf{T}[i]$ with two integers for the type of $\mathbf{T}_i$. If we have a bit array $\mathbf{hasS}$ of length σ that $\mathbf{hasS}[c] = true$ if a S-suffix starting with c appears in $\mathbf{T}$ and $\mathbf{hasS}[c] = false$ otherwise. For $1 \le i < N$, if $\mathbf{T}'[\mathbf{SA}[i]] = \mathbf{T}'[\mathbf{SA}[i+1]]$, $\mathbf{T}[\mathbf{SA}[i]] = \mathbf{T}[\mathbf{SA}[i+1]]$. Moreover $\mathbf{T}'[\mathbf{SA}[i]]$ and $\mathbf{T}'[\mathbf{SA}[i+1]]$ are L, S-suffix respectively and $\mathbf{hasS}[\mathbf{T}[\mathbf{SA}[i]]] = true$, $\mathbf{T}[\mathbf{SA}[i]] = \mathbf{T}[\mathbf{SA}[i+1]]$. By a left to right scan on $\mathbf{SA}$, we can determine the type of $\mathbf{T}_{\mathbf{SA}[i]}$ by comparing with $\mathbf{T}_{\mathbf{SA}[i]+1}$, and the character $\mathbf{T}[\mathbf{SA}[i]]$. Therefore we can restore $\mathbf{T}'$ from $\mathbf{T}$, $\mathbf{SA}$, and $\mathbf{hasS}$ in linear time.

7 Discussion

1. **T** を書き換えない、または **T** を in-place で復元可能な方法
2. ϕ 配列の in-place 計算

References

1 Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *Algorithms in Bioinformatics, Second International Workshop, WABI 2002, Rome, Italy, September 17-21, 2002, Proceedings*, pages 449–463, 2002. URL: `http://dx.doi.org/10.1007/3-540-45784-4_35`, `doi:10.1007/3-540-45784-4_35`.

2 M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

3 Jingchao Chen. Optimizing stable in-place merging. *Theor. Comput. Sci.*, 302(1-3):191–210, 2003. URL: `http://dx.doi.org/10.1016/S0304-3975(02)00775-2`, `doi:10.1016/S0304-3975(02)00775-2`.

4 Maxime Crochemore and Lucian Ilie. Computing longest previous factor in linear time and applications. *Inf. Process. Lett.*, 106(2):75–80, 2008. URL: `http://dx.doi.org/10.1016/j.ipl.2007.10.006`, `doi:10.1016/j.ipl.2007.10.006`.

5 Johannes Fischer, Volker Heun, and Stefan Kramer. Fast frequent string mining using suffix arrays. In *Proceedings of the 5th IEEE International Conference on Data Mining (ICDM 2005), 27-30 November 2005, Houston, Texas, USA*, pages 609–612, 2005. URL: `http://dx.doi.org/10.1109/ICDM.2005.62`, `doi:10.1109/ICDM.2005.62`.

6 Gianni Franceschini and S. Muthukrishnan. In-place suffix sorting. In *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings*, pages 533–545, 2007. URL: `http://dx.doi.org/10.1007/978-3-540-73420-8_47`, `doi:10.1007/978-3-540-73420-8_47`.

7 Keisuke Goto and Hideo Bannai. Space efficient linear time Lempel-Ziv factorization for small alphabets. In *Data Compression Conference, DCC 2014, Snowbird, UT, USA, 26-28 March, 2014*, pages 163–172, 2014. URL: `http://dx.doi.org/10.1109/DCC.2014.62`, `doi:10.1109/DCC.2014.62`.

8 Keisuke Goto, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Fast q-gram mining on SLP compressed strings. *J. Discrete Algorithms*, 18:89–99, 2013. URL: `http://dx.doi.org/10.1016/j.jda.2012.07.006`, `doi:10.1016/j.jda.2012.07.006`.

9 Guy Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 549–554, 1989. URL: `http://dx.doi.org/10.1109/SFCS.1989.63533`, `doi:10.1109/SFCS.1989.63533`.

10 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006. URL: `http://doi.acm.org/10.1145/1217856.1217858`, `doi:10.1145/1217856.1217858`.

11 Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001 Jerusalem, Israel, July 1-4, 2001 Proceedings*, pages 181–192, 2001. URL: `http://link.springer.de/link/service/series/0558/bibs/2089/20890181.htm`.

12 Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Constructing suffix arrays in linear time. *J. Discrete Algorithms*, 3(2-4):126–142, 2005. URL: `http://dx.doi.org/10.1016/j.jda.2004.08.019`, `doi:10.1016/j.jda.2004.08.019`.

13 Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*, 3(2-4):143–156, 2005. URL: `http://dx.doi.org/10.1016/j.jda.2004.08.002`, `doi:10.1016/j.jda.2004.08.002`.

14 Fugen Li and Gary D. Stormo. Selection of optimal DNA oligos for gene expression arrays. *Bioinformatics*, 17(11):1067–1076, 2001. URL: `http://dx.doi.org/10.1093/bioinformatics/17.11.1067`, `doi:10.1093/bioinformatics/17.11.1067`.

15 Zhize Li, Jian Li, and Hongwei Huo. Optimal in-place suffix sorting. In *Data Compression Conference, DCC 2018, Snowbird, UT, USA, 26-29 March, 2018*, 2018.

16 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. URL: `http://dx.doi.org/10.1137/0222058`, `doi:10.1137/0222058`.

17 Giovanni Manzini. Two space saving tricks for linear time LCP array computation. In *Algorithm Theory - SWAT 2004, 9th Scandinavian Workshop on Algorithm Theory, Humlebaek, Denmark, July 8-10, 2004, Proceedings*, pages 372–383, 2004. URL: `http://dx.doi.org/10.1007/978-3-540-27810-8_32`, `doi:10.1007/978-3-540-27810-8_32`.

18 Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001. URL: `http://doi.acm.org/10.1145/375360.375365`, `doi:10.1145/375360.375365`.

19 Ge Nong. Practical linear-time $O(1)$-workspace suffix sorting for constant alphabets. *ACM Trans. Inf. Syst.*, 31(3):15, 2013. URL: `http://doi.acm.org/10.1145/2493175.2493180`, `doi:10.1145/2493175.2493180`.

20 Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Computers*, 60(10):1471–1484, 2011. URL: `http://dx.doi.org/10.1109/TC.2010.188`, `doi:10.1109/TC.2010.188`.

21 Nicola Prezza. In-place sparse suffix sorting. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 1496–1508, 2018. URL: `https://doi.org/10.1137/1.9781611975031.98`, `doi:10.1137/1.9781611975031.98`.

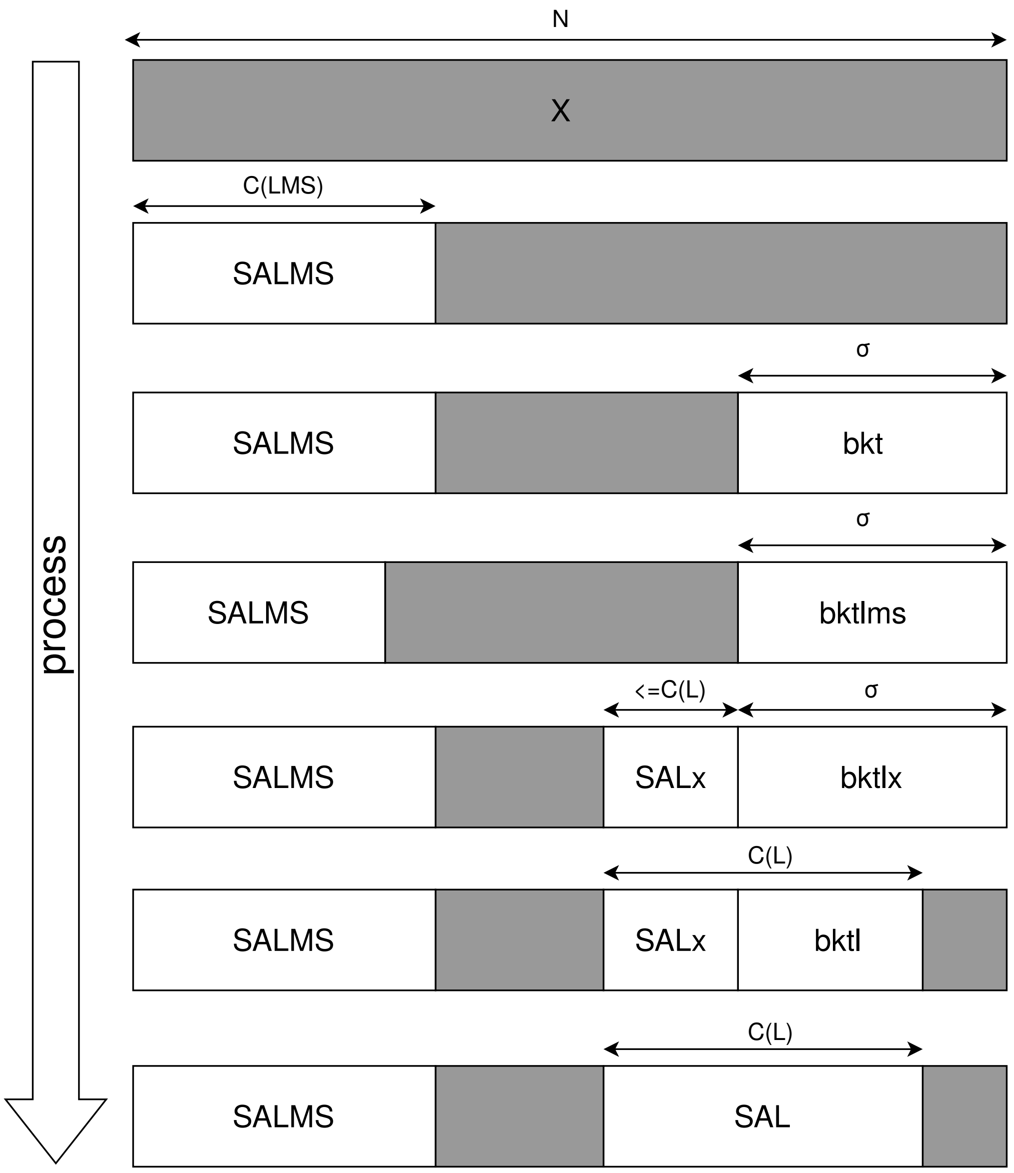

Figure 1 A transition of values of $\mathbf{X}$ to construct $\mathbf{SA}_L$ from $\mathbf{SA}_{suf(LMS)}$.

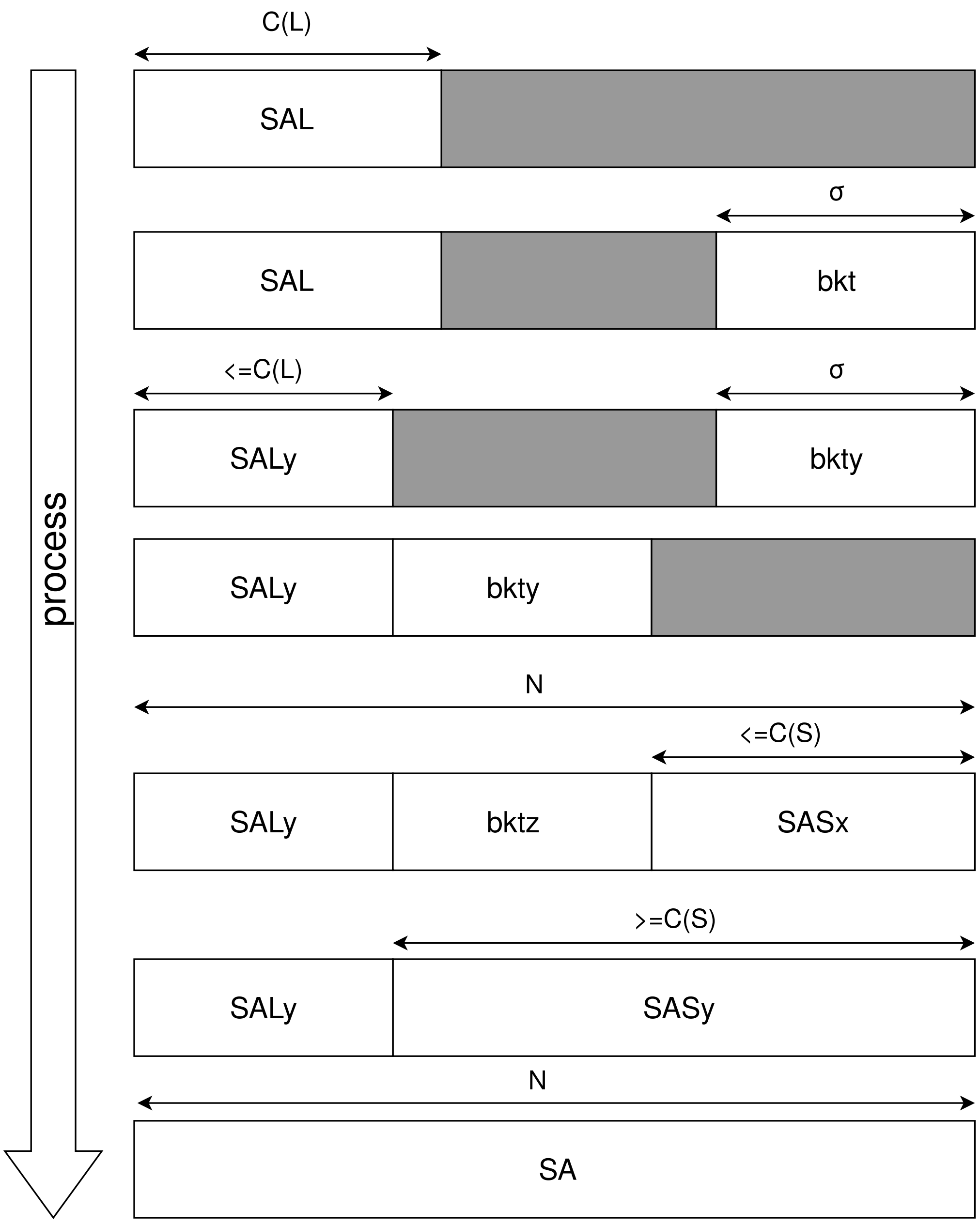

Figure 2 A transition of values of $\mathbf{X}$ to construct $\mathbf{SA}$ from $\mathbf{SA}_L$.