Optimal Lightweight Construction of Suffix Arrays for Constant Alphabets

Ge Nong[1,⋆] and Sen Zhang[2,⋆⋆]

[1] Computer Science Department,
Sun Yat-Sen University, GuangZhou 510275, PRC
issng@mail.sysu.edu.cn

[2] Department of Mathematics, Computer Science and Statistics,
SUNY College at Oneonta, NY 07104, U.S.A.
zhangs@oneonta.edu

Abstract. This article presents our divide-and-conquer optimal algorithms for lightweight suffix array construction for constant alphabets. These algorithms can efficiently compute the suffix array of a size-n text T with an alphabet Σ using $O(n \log \Sigma)$ time and $(\ell(T) + |\Sigma|\lceil \log n \rceil + O(1))$-bit *working space* (excluding the space for the output suffix array), where Σ is an integer or constant alphabet, and $\ell(T)$ is the length of T measured in bits. For popular applications in practice with $n \leq 2^{32}$ and $|\Sigma| \leq 256$, these results translate into $O(n)$ time and a total space of $5n + O(1)$ bytes, which are the optimal time and space complexities for lightweight suffix array construction.

1 Introduction

The suffix array is a fundamental index data structure used in a broad range of applications such as compression, string matching and computational biology [1]. For a n-character text T, its suffix array $SA(T)$ is an array of pointers for all suffixes in T sorted lexicographically, which requires $O(n\lceil \log n \rceil)$-bit space. The concept of suffix array was initially proposed by Manber and Myers in 1990 [2,3]. Since then, suffix arrays have been employed widely for data indexing, retrieving, storing and processing. For example, the Burrows-Wheeler transform [4] for building efficient compression solutions can be quickly computed by fast suffixes sorting based on suffix array construction. In many cases where suffix arrays are applied, constructing the suffix arrays generally constitutes the basis for subsequent tasks. Recently, it has been observed that the construction of suffix arrays is needed for large-scale applications where the input texts are huge with over billions characters (e.g., biology genome database) [5,6,7,8,9], which

⋆ Supported in part by the GuangDong Natural Science Foundation research grant number 06023193.

⋆⋆ Supported in part by the SUNY College Oneonta W.B. Ford Grants.

motivated the currently intensive research on time and space efficient suffix array construction algorithms (SACAs).

The suffix array is usually known as a space efficient alternative to the suffix tree. In general, applications built on suffix trees can run times faster than that using suffix arrays; however, the storage of a suffix tree also consumes a memory space times more than its suffix array counterpart. To maximize the benefit from using suffix arrays, it is highly desirable to further reduce the working space required for constructing a suffix array, where the term of *working space* in this context doesn't include the space for the output suffix array [1]. In 2002, Manzini and Ferragina [11] initially raised the problem of lightweight suffix array construction, which was informally termed as quickly constructing the suffix array of a size-n text using $5n + O(1)$ bytes, for $n \leq 2^{32}$ and the alphabet $|\Sigma| \leq 256$. One year later, Burkhardt and Kärkkäinen [10] presented their algorithm based on the concept of "difference covers", with $O(n \log n)$ worst-case runtime and using a working space as the input plus $O(n/\sqrt{\log n})$. In the same year, Hon and Sadakane et al. [7] showed that the suffix arrays can be constructed in optimal $O(n \log n)$ time and $O(n \log n)$-bit space for texts with integer alphabets, or $O(n)$ time and $O(n)$-bit working space for texts with constant alphabets. Later, Na [12] proposed an alphabet-independent linear $O(n)$ time algorithm for constructing suffix arrays using $O(n \log |\Sigma| \log^{\alpha}_{|\Sigma|} n)$-bit working space, where $\alpha = \log_3 2$. Very recently, Puglisi and Smyth et al. [1] conducted a thorough survey on SACAs including [11,10,7,13,14,15,16,17,18,19,20,12], and concluded that to devise an optimal SACA which is *fast*, *lightweight* and *linear* in the worst case still remains as a challenge.

What of our particular interest here is the optimal lightweight suffix array construction for texts with constant alphabets, which we term as to construct suffix arrays with the known optimal time complexities, and meanwhile, using a space as small as possible. Specifically, we present here our novel solution with a set of practical algorithms for optimal lightweight suffix array construction for constant alphabets, which is optimal in the sense that it can compute the suffix array of a size-n text of $n \log |\Sigma|$-bit using $O(n \log |\Sigma|)$ time and $(n \log |\Sigma| + |\Sigma| \log n + O(1))$-bit working space (excluding the space for the output suffix array), where Σ is an integer or constant alphabet. For popular applications in practice with $n \leq 2^{32}$ and $|\Sigma| \leq 256$, these results translate into $O(n)$ time and a total space of $5n + O(1)$ bytes, which are the optimal time and space complexities for *lightweight* suffix array construction termed by Manzini and Ferragina [11]. We intentionally don't use the big-O notations for n and $|\Sigma|$ in the space complexity formulas, in order to show the accurate space requirement which is a main concern for lightweight suffix array construction.

The rest of this article is organized as following. Section 2 introduces the preliminaries including some general notations and assumptions. Our solution for optimal lightweight suffix array construction is presented in Section 3. Section 4 summarizes the main results. Finally, Section 5 gives the conclusion.

[1] In the literature for suffix array construction algorithms, the working space may exclude both the input text and the output suffix array, for example, in [10].

2 Preliminaries

2.1 Notations

Let $T = t_0t_1t_2 \dots t_{n-2}\$$ be the input text with n characters arranged as an array, where the characters are in an alphabet Σ. Two kinds of alphabets are considered for Σ here: (1) an integer alphabet with characters in the range of $[0, n^{O(1)})$, and (2) a constant alphabet of size $O(1)$. Without loss of generality, the characters of T, from left to right, are indexed starting from 0. The last character \$ is called the *sentinel*, which is unique in T, not in Σ and lexicographically smaller than any character in Σ.

For a size-m text $X = x_0x_1x_2 \dots x_{m-2}\$$, we define some notations as below:

- $S(X, i)$: the suffix in X starting at x_i and running to the sentinel \$, i.e. $S(X, i) = x_ix_{i+1} \dots x_{m-2}\$$, where $i \in [0, m)$.
- $SA(X)$: the suffix array of X, which is the pointer array for all the m suffixes in X sorted in their lexicographically order, i.e., each item in $SA(X)$ contains an unique pointer to a suffix in X. Without loss of generality, the sorted order is assumed to be ascending.
- $ISA(X)$: the inverse suffix array of X, defined as $SA[ISA[i]] = i$, for $i \in [0, m)$.

Let $\prec$ and $\succ$ be the lexicographical preceding and succeeding operators, respectively, we define $\tau(T, i)$ to be the LS-type function for T, given as:

$$\tau(T,i) = \begin{cases} 0, & for\ S(T,i) \succ S(T,i+1)\ and\ i \in [0,n); \\ 1, & for\ (S(T,i) \prec S(T,i+1)\ and\ i \in [0,n-1))\ or\ i = n-1. \end{cases}$$

For denotation simplicity, a character $T[i]$ is said to be type-L or type-S for $\tau(T, i) = 0$ or 1, respectively. Moreover, a suffix $S(T, i)$ is said to be a type-L or type-S suffix if $T[i]$ is type-L or type-S, respectively. Let B be the size-n array $[0..n-1]$ of integers allocated for storing the output suffix array $SA(T)$, in which each item is $\lceil \log n \rceil$-bit. Next, $B[i]$ and $T[i]$ are said to be a pair of siblings, and an item $B[i]$ is said to be type-L or type-S if its sibling $T[i]$ is type-L or type-S, respectively. Further, let B_S and B_L be two sets consisting of the last n_0 type-S and type-L items in B, respectively, where $n_0 = |T_0| \leq \lceil n/2 \rceil$ and T_0 will be defined in Section 3.1. To denote a substring $x_ix_{i+1} \dots x_j$ in a text X, a simpler form of $X[i, \dots, j]$ could be used. In addition, we use $\ell(\upsilon_0, \dots, \upsilon_k)$ to denote the total length of all objects υ_0, ..., υ_k measured in bits.

2.2 Assumptions

Unless otherwise specified, in this article, we have the following general assumptions:

- The *working space* of a computation doesn't include the space for the output. In other words, the working space equals to the total space minus the space for the output. Without explicit specification, the term *space* implies the total space.

– The space is provided in a unit-cost RAM with word size $O(\log W)$-bit, where $n \leq W$. Following this assumption, a standard arithmetic or bitwise boolean operation on word-sized operands costs $O(1)$ time.

3 Solution

Our divide-and-conquer solution for computing $SA(T)$ for T is presented in this section. To solve the problem at hand, we first reduce the problem, next compute the suffix array for the reduced problem and then based on which, to derive the suffix array for the original problem.

3.1 Reducing the Problem

First of all, we introduce some basic definitions for reducing the problem. A S-string in T is: (1) $t_i \dots t_j$ $(0 \leq i \leq j < n)$ for both t_i and t_j are type-S; or (2) the sentinel \$ itself. In addition, the rank of t_i is defined as the number of characters less than t_i in T; all the ranks of T starts from 0. Further, let $Z_k(T, i)$ denote a substring in T consisting of the k $(k > 0)$ consecutive S-strings starting at a type-S character $T[i]$, in which fewer S-strings are possible when the sentinel \$ is included. Any k consecutive S-strings in T is called a Z_k-string.

Let T_0 be the text consists of the ranks (also known as the lexicographical names) of all S-strings in T, and let $n_0 = |T_0|$. The Corollary 2 in [16] says that the sorted order of all suffixes in T_0 determines the sorted order of all type-S suffixes in T. Further, let % denote the modulo operator and T_1 be the text consisting of the ranks for triples in $\{T_0[i, i+1, i+2] : i\%3 \neq 2, i \in [0, n_0)\}$, where the ones for $\{i\%3 = 0\}$ are arranged in one consecutive block following by another block consisting of those for $\{i\%3 = 1\}$, and let $n_1 = |T_1|$. Similarly, let T_2 be the text consisting of the ranks for triples in $\{T_0[i, i+1, i+2] : i \bmod 3 = 2, i \in [0, n_0)\}$; and let $n_2 = |T_2|$. Without loss of generality, we assume that there are less [2] type-S characters than the type-L characters in T and n is even for presentation simplicity, which leads to $n_1 \leq \lceil 2n_0/3 \rceil \leq \lceil n/3 \rceil$. Although T_1 can be computed by first computing T_0 from T and then computing T_1 from T_0, doing in this way is too space consuming for our purpose. In the following, we design a space efficient algorithm for directly computing T_1 from T.

To present the algorithm, we continue to introduce some more definitions. Let P_1 be the index array for all ith Z_3-strings of T satisfying $i\%3 \neq 2$, i.e., $P_1[i]$ gives the pointers to all these ith Z_3-strings in T; specifically, in P_1, the ones for $\{i\%3 = 0\}$ and $\{i\%3 = 1\}$ are arranged in two consecutive blocks, respectively. Next, let P_1' be the result array of sorting all elements in P_1 in the lexicographically ascending order of their corresponding Z_3-strings in T, where ties between any two Z_3-strings with different lengths are broken by giving the

[2] In case that the type-S characters are more, the same discussion can be conducted symmetrically on the type-L characters, see [16] for details.

shorter a higher priority [3]. Further, let H_1 be an array for recording the lengths of all Z_3-strings in P_1. From the definitions of P_1, P_1' and H_1, each of them is an array with n_1 items in the range of $[0, n)$, which means that each item can be encoded in $\lceil \log n \rceil$-bit.

Because $n_1 \leq \lceil 2n_0/3 \rceil \leq \lceil n/3 \rceil$, instead of constructing $SA(T)$ directly, we can compute $SA(T_1)$ and then derive $SA(T)$ from $SA(T_1)$ using the algorithms developed below. The whole procedure is started by computing T_1 from T. In brief, computing T_1 from T consists of three steps: (1) compute P_1 from T; (2) compute P_1' from P_1 and T; and (3) compute T_1 from P_1' and T. We further to go through them one-by-one.

First, we have the below lemma for determining the type of a character in T.

Lemma 1. *For $i \in [0, n-1)$, we have (1) $\tau(T, i) = 0$ if $T[i] > T[i+1]$; (2) $\tau(T, i) = 1$ if $T[i] < T[i+1]$; and (3) $\tau(T, i) = \tau(T, i+1)$ if $T[i] = T[i+1]$.*

Proof. The correctness of (1) and (2) is obvious from the definition of $\tau(T, i)$. For (3), if $\tau(T, i+1) = 0$ or 1, we have $S(T, i+1) \succ S(T, i+2)$ or $S(T, i+1) \prec S(T, i+2)$, respectively. Given $T[i] = T[i+1]$, this yields $S(T, i) \succ S(T, i+1)$ or $S(T, i) \prec S(T, i+1)$, respectively, i.e. $\tau(T, i) = \tau(T, i+1)$.

We proceed to compute P_1 and then sort P_1 to obtain P_1'.

Lemma 2. *Given T, P_1 can be computed using $O(n)$ time and a working space of $\ell(T) + O(1)$ bits.*

Proof. This can be done by simply traversing T once from right to left, to record in P_1 the positions of all type-S characters in T. At each step, the type of the current character is derived from that of its immediately succeeding character in $O(1)$ time, using Lemma 1.

Lemma 3. *(time bottleneck) Given T, P_1 and H_1, P_1' can be computed using $O(\ell(T))$ time and a total space of at most $\ell(B, T) + O(1)$ bits.*

Proof. Omitted due to the space limit.

For computing T_1 from P_1', all the Z_3-strings in T need to be sorted. For which, we have the below lemma for retrieving a S-string from its head in T.

Lemma 4. *Given $T[i]$ is type-S, starting from $T[i]$, we can find the first type-S item $T[j]$ succeeding to $T[i]$ by traversing up to the first type-L item $T[k]$ succeeding to $T[j]$, where $i < j < k$.*

Proof. If $T[i] = T[i+1]$, we know that $T[i+1]$ must be type-S and $j = i+1$. Suppose $T[i] < T[i+1]$, starting from $T[i+1]$, we traverse forward to the first $T[k]$ ($k > i+1$) satisfying $T[k-1] < T[k]$. At this moment, we know that $T[k-1]$ must be type-S. From $T[k]$, we traverse backward to the first $T[j]$ satisfying $T[j-1] > T[j]$ ($j < k$).

[3] The correctness of this tie-breaking scheme is supported by the Lemma 2 in [16], which states that if $T[i] = T[j]$, $T[i]$ is type-L and $T[j]$ is type-S, then $S(T, i) \prec S(T, j)$.

Without the space constraint, T_1 can be easily computed from P'_1 and T using an auxiliary array as large as B. We design a space efficient algorithm Compute-T1 to avoid using such a large auxiliary array, from which we have the below result, which proof is omitted due to the space limit.

Lemma 5. *Given P'_1 and T, T_1 can be computed using $O(n)$ time and a total space of $\ell(B,T)+O(1)$ bits.*

Hence, we have the following result on the complexities for computing T_1 from T.

Lemma 6. *Given T, T_1 can be computed using $O(\ell(T))$ time and a total space of $\ell(T)+3\ell(T_1)+O(1)$ bits.*

Proof. The time and space complexities for the three steps for computing T_1 from T are given by Lemma 2, 3 and 5, respectively. Both the maximum time and the maximum space are observed for computing T_1 from P'_1, H_1 and T, which dominate the total time and space complexities.

Now, we have successfully reduced the problem size from n to $\lceil n/3 \rceil$. We proceed to solve the reduced problem, i.e., to compute $SA(T_1)$ from T_1.

3.2 Solving the Reduced Problems

We design in Fig. 1 the algorithm Lightweight-KS-Sort for computing $SA(T_1)$ from T_1 in $O(n)$ time and $\ell(B)$ space, which is a lightweight alternative to the traditional KS algorithm and described as follows:

- Let U, V and X be 3 arrays $[0, m-1]$ of integers, where each item is $\lceil \log m \rceil$-bit. Moreover, suppose the buffers for U, V are allocated consecutively. Let SA_x denote $SA(X)$.
- Let X_1 and X_2 denote the texts consisting of the lexicographical names for triples in $\{X[i,i+1,i+2] : i\%3 \neq 2, i \in [0,m)\}$ and $\{X[i,i+1,i+2] : i\%3 = 2, i \in [0,m)\}$, respectively (in X_1, the names for triples with $\{i\%3 = 0\}$ are in a consecutive block and those for $\{i\%3 = 1\}$ are in another.), and $X_0 = X_1 \oplus X_2$, where "$\oplus$" denotes the text concatenating operator. Moreover, let $m_1 = |X_1|$ and $m_2 = |X_2|$. We first to compute X_1 and X_2 from X into V, using bucket sorting with U as the counter array and V as the bucket array. Then, X_1 and X_2 are copied to X for later use. This step can be done in $O(m)$ time.
- (Now, there are two copies of X_1, one in the last m_1 items of V, and another in X.) Let V_1 denote the size-m_1 array immediately right to the X_1 in V, and U_1 be the size-m_1 array immediately right to V_1. We make the function call Lightweight-KS-Sort(X_1, U_1, V_1) to recursively compute SA_{x1}, where SA_{x1} denotes $SA(X_1)$. This step can be done in $O(m)$ time.
- Provided with SA_{x1} in V and the X_2's copy in X, we use the induced sorting method in the KS algorithm to compute SA_{x2} from SA_{x1} and X_2, where SA_{x2} denotes $SA(X_2)$. This step can be done in $O(m)$ time. As the result, SA_{x1} and SA_{x2} are stored in V.

- Let SA_{x0} denote $SA(X_0)$. Now, we are going to merge SA_{x1} and SA_{x2} to produce SA_{x0}. From SA_{x1} and SA_{x2} in V, we compute ISA_{x1} and ISA_{x2} into U, where ISA_{x1} and ISA_{x2} are the inverse SA_{x1} and SA_{x2}, defined as $SA_{x1}[ISA_{x1}[i]] = i$ ($i \in [0, m_1)$) and $SA_{x2}[ISA_{x2}[j]] = j$ ($j \in [0, m_2)$). This can be done in $O(m)$ time.
- Let Y_1 and Y_2 be defined as $SA_{x1}[i] = SA_{x0}[Y_1[i]]$ and $SA_{x2}[j] = SA_{x0}[Y_2[j]]$, respectively, for $i \in [0, m_1)$ and $j \in [0, m_2)$. Y_1 and Y_2 are called the position index arrays for SA_{x1} and SA_{x2}, which give the position indices for all items in SA_{x1} and SA_{x2} in the merged suffix array SA_{x0}. Computing Y_1 and Y_2 can be done by traversing SA_{x1} and SA_{x2} once and using the buffers for SA_{x1} and SA_{x2} only, i.e., V is updated with Y_1 and Y_2. This can be done in $O(m)$ time.
- Now, U contains ISA_{x1} and ISA_{x2}, and V contains Y_1 and Y_2. To compute ISA_{x0}, which is the inverse SA_{x0}, we simply traverse U once to set $U[i] = V[U[i]]$, for each $i \in [0, m)$. From the definitions of ISA_{x1}, ISA_{x2}, Y_1 and Y_2, it is trivially to see that U contains ISA_{x0} now. This can be done in $O(m)$ time.
- Given ISA_{x0} in U, we compute SA_{x0} into V by traversing U once to set $V[U[i]] = i$, for each $i \in [0, m)$. This can be done in $O(m)$ time.
- Now, because the original positions of the elements of X_1 and X_2 in X are interleaved (every two elements of X_1 followed by an element of X_2), we traverse SA_{x_0} once with a complexity of $O(m)$ to compute SA_x as

$$SA_x[i] = \begin{cases} 3SA_{x_0}[i], \ for \ SA_{x_0}[i] \in [0, \lceil m_1/2 \rceil); \\ 3(SA_{x_0}[i] - \lceil m_1/2 \rceil) + 1, \ for \ SA_{x_0}[i] \in [\lceil m_1/2 \rceil, m_1); \\ 3(SA_{x_0}[i] - m_1) + 2, \ for \ SA_{x_0}[i] \in [m_1, m). \end{cases} \quad (1)$$

- Finally, we copy SA_x in V to X for returning the result.

Lemma 7. *Given a size-n text X with each character encoded in $\lceil \log n \rceil$-bit, $SA(X)$ can be computed in $O(n)$ time and using a total space of $3\ell(X) + O(1)$ bits.*

Proof. In the LIGHTWEIGHT-KS-SORT algorithm, The time is governed by the recurrence $T(n) = T(\lceil 2n/3 \rceil) + O(n)$ and $T(n) = O(1)$ for $n < 3$, which leads to $T(n) = O(n)$. At each iteration, when making the recursion call in line **??**, only X is occupied, and U and V are available for use as the buffer space for the recursion. Hence, (omitting the space used for the recursion stack which is $O(\log n)$-bit and commonly neglected in the literature for suffix array construction algorithms) a total space of $\ell(X, U, V) = 3\ell(X) + O(1)$ bits is sufficient for the recurrence.

Recalling that each element of T_1 is $\lceil \log n_1 \rceil$-bit and $n_1 \leq \lceil n/3 \rceil$, Lemma 7 immediately suggests the following result.

Corollary 1. *Given T_1, $SA(T_1)$ can be computed using $O(n)$ time and a total space of $3\ell(T_1) + O(1)$ bits.*

Provided that $SA(T_1)$ is know, it is only a routine job for us to induce $SA(T_2)$ from $SA(T_1)$ using the KS skew algorithm [15], in two steps as follows:

- Compute T_2 from T, using the method for computing T_1. This step can be done using $O(\ell(T))$ time and a total space of $\ell(T) + 3\ell(T_2) + O(1)$ bits (see Lemma 6).
- Compute $SA(T_2)$ from $SA(T_1)$ and T_2, using the KS skew algorithm. This can be done using $O(n)$ time and a total space of $\ell(T_1) + 3\ell(T_2) + O(1)$ bits.

```
LIGHTWEIGHT-KS-SORT($X$, $U$, $V$)
    $\triangleright$ Input: $X$—array $[0..m-1]$ of integer, each item is $\lceil \log m \rceil$-bit.
    $\triangleright$ Output: $SA_x$—the suffix array of $X$, returned in the buffer of $X$.
    $\triangleright$ $U$, $V$: array $[0..m-1]$ of integer, each item is $\lceil \log m \rceil$-bit.
    $\triangleright$ Assumption: The buffers for $U$ and $V$ are allocated consecutively .
 1  if $|X| < 3$
 2      then Directly compute $SA_x$ from $X$ and store the result in $X$.
 3           return
 4  Compute $X_1$ and $X_2$ from $X$ into $V$. $\triangleright$ $X_1$ is stored in the last $m_1$ items of $V$.
 5  Copy $X_1$ and $X_2$ from $V$ to $X$. $\triangleright$ Save $X_1$ and $X_2$ for computing $SA_{x2}$ later.
 6  $m_1 \leftarrow |X_1|$; $m_2 \leftarrow |X_2|$
 7  Let $U_1$ and $V_1$ be the two arrays $[0..m_1-1]$ immediately right to the $X_1$ in $V$.
 8  LIGHTWEIGHT-KS-SORT($X_1, U_1, V_1$) $\triangleright$ Compute $SA_{x1}$ into $V_1$.
 9  Induced sorting $SA_{x2}$ from $SA_{x1}$ into $U_1$.
10  Traverse $SA_{x1}$ and $SA_{x2}$ in $V$ once to compute $ISA_{x1}$ and $ISA_{x2}$ into $U$.
    $\triangleright$ Now, $U$ contains $[ISA_{x1}, ISA_{x2}]$; $V$ contains $[SA_{x1}, SA_{x2}]$; $X$ contains $[X_1, X_2]$.
11  Traverse $SA_{x1}$ and $SA_{x2}$ in $V$ once to update $V$ with the position index arrays
    $Y_1$ and $Y_2$.
12  For each $i \in [0, m)$, $U[i] \leftarrow V[U[i]]$. $\triangleright$ Compute $ISA_{x0}$.
13  For each $i \in [0, m)$, $V[U[i]] \leftarrow i$. $\triangleright$ Compute $SA_{x0}$.
14  For each $i \in [0, m)$, compute $SA_x$ from $SA_{x0}$ by Eq.(1).
15  Copy $V$ to $X$. $\triangleright$ Return the result in $X$.
16  return
```

Fig. 1. The linear lightweight KS sorting algorithm

Recalling that $n_1 \leq n/3$, $n_2 \leq n/6$, and B, T_1 and T_2 have the same item's size of $\lceil \log n \rceil$-bit, the maximum space of the above two-step procedure is upper bounded by $\ell(T, B) + O(1)$ bits. The complexities for inducing $SA(T_2)$ from $SA(T_1)$ is concluded as follows.

Lemma 8. *Given $SA(T_1)$ and T, $SA(T_2)$ can be computed using $O(\ell(T))$ time and a total space of $\ell(T, B) + O(1)$ bits.*

3.3 Inducing the Final Result

Having solved $SA(T_1)$ and $SA(T_2)$, we proceed to merge $SA(T_1)$ and $SA(T_2)$ into $SA(T_0)$ in $O(n)$ time and $(\ell(T, B) + O(1))$-bit space. If we use the skew method in the LIGHTWEIGHT-KS-SORT algorithm, a total space of $3\ell(T_1, T_2) + O(1) =$

$1.5\ell(B) + O(1)$ bits is required, which is too space consuming. Notice that in the algorithm Lightweight-KS-Sort, X is used only for rank comparisons. Given T, $SA(T_1)$ and SA_2, we design a more space efficient algorithm for this job, which performs rank comparisons using Z_3-strings in T. This algorithm is described below:

- Let SA'_1 and SA'_2 be defined as $SA'_1 = \{P_1[SA(T_1)[i]] : i \in [0, n_1)\}$ and $SA'_2 = \{P_2[SA(T_2)[i]] : i \in [0, n_2)]$, respectively. An item $B[i]$ is said to in a set X if $B[i]$ is allocated for storing an item in X. Further, supposed that SA'_1 and SA'_2 are initially stored in the first n_0 (recalling that $n_0 = n_1 + n_2$) items of B, and SA'_1 is left to SA'_2.
- First, we move SA'_1 and SA'_2 into the type-L items in B_L by traversing B once from right to left. At this step, we record in h the position of the first $B[i]$ in SA'_2. This can be done in $O(n)$ time.
- Next, ISA'_1 and ISA'_2 are computed into the type-S items in B_S, where ISA'_1 is defined as for each $B[i]$ in SA'_1, $ISA'_1[B[i]] = i$, and similarly, ISA'_2 is defined as for each $B[i]$ in SA'_2, $ISA'_2[B[i]] = i$, where $i \in [0, n)$.
- Now, B_L contains SA'_1 and SA'_2, and B_S contains ISA'_1 and ISA'_2. We proceed to merge-sort SA'_1 and SA'_2. We first traverse B_L to compute the position index of each item in SA'_1 and SA'_2 in the merged set SA_0, where SA_0 denotes $SA(T_0)$ and the index starts from 0. To determine the order between an item $B[u]$ in SA'_1 and an item $B[v]$ in SA'_2, let $i = B[u]$ and $j = B[v]$, we compare $Z_3(T, i)$ and $Z_3(T, j)$ (these two strings can be retrieved utilizing Lemma 4). If they are different, the order is immediately determined; or else we continue to compare as follows [4]:
  - Suppose $Z_3(T, i)$ is the kth Z_3-string in T, $k \in [0, n_0)$, we say $Z_3(T, i)$ is a residue-0 or residue-1 string if $k \mod 3 = 0$ or 1, respectively.
  - In case that $Z_3(T, i)$ is a residue-0 string, compare $Z_3(T, i')$ with $Z_3(T, j')$, where $Z_3(T, i')$ and $Z_3(T, j')$ are the first S-strings succeeding to $Z_3(T, i)$ and $Z_3(T, j)$, respectively. The order of $Z_3(T, i')$ and $Z_3(T, j')$ is determined by $B[i']$ and $B[j']$, for ISA'_1 and ISA'_2 are stored in B_S and both $B[i']$ and $B[j']$ are in B_S.
  - In case that $Z_3(T, i)$ is a residue-1 string, compare $Z_3(T, i'')$ with $Z_3(T, j'')$, where $Z_3(T, i'')$ and $Z_3(T, j'')$ are the 2nd S-strings succeeding to $Z_3(T, i)$ and $Z_3(T, j)$, respectively. Similar to the previous case, the order of $Z_3(T, i'')$ and $Z_3(T, j'')$ is determined by $B[i'']$ and $B[j'']$, for both $B[i'']$ and $B[j'']$ are in B_S.

  Retrieving the strings, once more, can be done utilizing Lemma 4. Checking if $Z_3(T, i)$ is a residue-0 or residue-1 string can be done in $O(1)$ time by simply comparing $B[i]$ with h, for all items of SA'_1 are stored before the item $B[h]$. Because each S-string in T can be visited at most four times (1 due to locating the terminating character of the S-string preceding to it and 3 due to S-string comparisons) for merging SA_1 and SA_2, this step is done in $O(n)$ time.

[4] The correctness of this comparison scheme can be trivially seen from the KS skew algorithm in [15].

– Let ISA_0 denote the inverse $SA(T_0)$. ISA_0 is computed by first traversing B once from right to left to set each type-S item $B[i]$ with $B[B[i]]$; and then traverse B once more from right to left to move all the items in B_S into the last n_0 items of B. This can be done in $O(n)$ time.
– Given ISA_0 (in the last n_0 items of B), SA_0 can be easily computed into the left half of B in $O(n)$ time.

As a summary for the above algorithm, we have the following lemma.

Lemma 9. *Given T, $SA(T_1)$ and $SA(T_2)$, $SA(T_0)$ can be computed using $O(n)$ time and a total space of $\ell(B,T) + O(1)$ bits.*

Given SA_0 and T, we can compute $SA(T)$ in two steps, as described below:

– Let P_0 be the S-string array for T, which is the index array for all S-strings in T. In addition, let SA_S denote the suffix array for all type-S suffixes in T, defined as $SA_S = \{SA(T)[i] : T[SA(T)[i]]\ is\ type-S, i \in [0,n)\}$. Given SA_0 (stored in the left half of B), SA_S can be computed in two steps: (1) compute P_0 into the right half of B by traversing B once from right to left; and (2) compute SA_S by traversing SA_0 once to set $SA_0[i] = P_0[SA_0[i]]$. Now, SA_S is contained in the first n_0 items of B. This step can be done in $O(n)$ time.
– Use the KA skew algorithm [16] to induce $SA(T)$ from SA_S and T, which requires a bucket counter array [5] of $|\Sigma|\lceil\log n\rceil$-bit in addition to B and T, and is done in $O(n)$ time. The maximum total space for the whole procedure of computing $SA(T)$ from T is due to this step.

Hence, the complexities for computing $SA(T)$ from $SA(T_0)$ are concluded as follows.

Lemma 10. *(space bottleneck) Given T and $SA(T_0)$, $SA(T)$ can be computed using $O(n)$ time and a total space of $\ell(B,T) + |\Sigma|\lceil\log n\rceil + O(1)$ bits.*

4 Main Results

Theorem 1. *For a size-n text T with an integer or constant alphabet Σ, $SA(T)$ can be computed using $O(n \log|\Sigma|)$ time and a total space of $\ell(T,B)+|\Sigma|\lceil\log n\rceil+ O(1)$ bits.*

Proof. The whole procedure for computing $SA(T)$ from T consists of the following steps in sequence:

1. Compute T_1 from T and $SA(T_1)$ from T_1, see Lemma 6 and Corollary1.
2. Compute $SA(T_2)$ from $SA(T_1)$ and T, see Lemma 8.
3. Merge $SA(T_1)$ and $SA(T_2)$ into $SA(T_0)$, see Lemma 9.
4. Induce $SA(T)$ from $SA(T_0)$, see Lemma 10.

[5] Two bucket counter arrays can be used for higher speed when $|\Sigma|$ is not large, e.g., $O(1)$.

The bucket sorting function for computing P'_1 from P_1 and T in Step 1, as well as that for computing P'_2 from P_2 and T in Step 2, dominate the whole procedure's time complexity, which is $O(\ell(T)) = O(n \log |\Sigma|)$ from Lemma 3.

The total space consists of 3 parts: (1) the array T for the input text; (2) the array B for the output suffix array; and (3) the $|\Sigma|\lceil \log n \rceil$-bit bucket counter array for inducing $SA(T)$ from $SA(T_0)$.

From Theorem 1, we have the following two results for texts with integer and constant alphabets, respectively.

Corollary 2. *For a size-n text T with an integer alphabet Σ, where $|\Sigma| \leq n$, $SA(T)$ can be computed using $O(n \log n)$ time and a total space of at most $3n\lceil \log n \rceil + O(1)$ bits.*

Corollary 3. *(optimal lightweight) For a size-n text T with a constant alphabet Σ, where $n \leq 2^{32}$ and $|\Sigma| \leq 256$, $SA(T)$ can be computed using $O(n)$ time and a total space of $5n + O(1)$ bytes.*

5 Conclusion

A divide-and-conquer solution with a set of practical algorithms has been developed in this work for optimal lightweight suffix array construction. The crucial task for developing this solution, as we have shown, is how to reduce the problem size to be small enough so that the reduced problem can be efficiently computed and meanwhile, the final suffix array can be augmented from the reduced one time and space efficiently. Once the problem has been reduced, a number of traditional suffix array construction algorithms are allowed to be further exploited to solve the reduced problem. This makes the solution flexible to be further improved for better average performance.

Acknowledgment

The authors wish to thank the anonymous reviewers for their constructive suggestions and insightful comments.

References

1. Puglisi, S.J., Smyth, W.F., Turpin, A.: A taxonomy of suffix array construction algorithms. ACM Computing Surveys 2006 (to Appear)
2. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. In: Proceedings of the first ACM-SIAM Symposium on Discrete Algorithms, pp. 319–327 (1990)
3. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. SIAM Journal on Computing 22(5), 935–948 (1993)
4. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Technical Report SRC Research Report 124, Digital Systems Research Center, California, USA (1994)

5. Manzini, G., Ferragina, P.: Engineering a lightweight suffix array construction algorithm. Algorithmica 40(1), 33–50 (2004)
6. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In: Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC'00), pp. 397–406 (2000)
7. Hon, W.K., Sadakane, K., Sung, W.K.: Breaking a time-and-space barrier for constructing full-text indices. In: Proceedings of FOCS'03, pp. 251–260 (2003)
8. Lam, T.W., Sadakane, K., Sung, W.K., Yiu, S.M.: A space and time efficent algorithm for constructing compressed suffix arrays. In: Proceedings of International Conference on Computing and Combinatorics, pp. 401–410 (2002)
9. Kurtz, S.: Reducing the space requirement of suffix trees. Software Practice and Experience 29, 1149–1171 (1999)
10. Burkhardt, S., Kärkkäinen, J.: Fast lightweight suffix array construction and checking. In: Baeza-Yates, R.A., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 55–69. Springer, Heidelberg (2003)
11. Manzini, G., Ferragina, P.: Engineering a lightweight suffix array construction algorithm. In: Möhring, R.H., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 698–710. Springer, Heidelberg (2002)
12. Na, J.C.: Linear-time construction of compressed suffix arrays using O(nlog n)-bit working space for large alphabets. In: Apostolico, A., Crochemore, M., Park, K. (eds.) CPM 2005. LNCS, vol. 3537, pp. 57–67. Springer, Heidelberg (2005)
13. Larsson, N.J., Sadakane, K.: Faster suffix sorting. Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1–20/(1999), Department of Computer Science, Lund University, Sweden (1999)
14. Maniscalco, M.A., Puglisi, S.J.: Faster lightweight suffix array construction. In: Proceedings of 17th Australasian Workshop on Combinatorial Algorithms (AWOCA'06), pp. 16–29 (2006)
15. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. Journal of the ACM (6), 918–936 (2006)
16. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. In: Baeza-Yates, R.A., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 200–210. Springer, Heidelberg (2003)
17. 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)
18. Itoh, H., Tanaka, H.: An efficient method for in memory construction of suffix arrays. In: Proceedings of String Processing and Information Retrieval Symposium (1999)
19. Seward, J.: On the performance of BWT sorting algorithms. In: Proceedings DCC 2000 Data Compression Conference, Snowbird, UT, USA, pp. 173–82 (2000)
20. Schürmann, K.B., Stoye, J.: An incomplex algorithm for fast suffix array construction. In: Proceedings of 7th Workshop on Algorithm Engineering and Experiments (ALENEX/ANALCO 2005), pp. 77–85 (2005)