Linear-Time Construction of Compressed Suffix Arrays Using $o(n \log n)$-Bit Working Space for Large Alphabets[⋆]

Joong Chae Na

School of Computer Science and Engineering, Seoul National University
jcna@theory.snu.ac.kr

Abstract. The *suffix array* is a fundamental index data structure in string algorithms and bioinformatics, and the *compressed suffix array (CSA)* and the *FM-index* are its compressed versions. Many algorithms for constructing these index data structures have been developed. Recently, Hon et al. [11] proposed a construction algorithm using $O(n \cdot \log\log|\Sigma|)$ time and $O(n \log|\Sigma|)$-bit working space, which is the fastest algorithm using $O(n \log|\Sigma|)$-bit working space.
In this paper we give an efficient algorithm to construct the index data structures for large alphabets. Our algorithm constructs the suffix array, the CSA, and the FM-index using $O(n)$ time and $O(n \log|\Sigma| \log^{\alpha}_{|\Sigma|} n)$-bit working space, where $\alpha = \log_3 2$. Our algorithm takes less time and more space than Hon et al.'s algorithm. Our algorithm uses least working space among alphabet-independent linear-time algorithms.

1 Introduction

Given a string T of length n over an alphabet Σ, the suffix array due to Manber and Myers [16] and independently due to Gonnet et al. [6] is basically a sorted list of all the suffixes of T. The suffix array requires $O(n \log n)$-bit space. Manber and Myers [16] and Gusfield [9] proposed $O(n \log n)$-time algorithms for constructing the suffix array. Recently, almost at the same time, Kim et al. [13], Kärkkäinen and Sanders [12], and Ko and Aluru [14] developed algorithms to directly construct the suffix array in $O(n)$ time and $O(n \log n)$-bit space. These algorithms are based on similar recursive divide-and-conquer schemes.

As the size of data such as DNA sequences increases, compressed versions of the suffix array such as the *compressed suffix array (CSA)* [7, 8] and the *FM-index* [5] were proposed to reduce the space requirement of suffix arrays. Lam et al. [15] first developed an algorithm to directly construct the CSA, which uses $O(|\Sigma| n \log n)$ time and $O(n \log|\Sigma|)$ bits. Hon et al. [10] improved the construction time to $O(n \log n)$, while maintaining the $O(n \log|\Sigma|)$-bit space complexity.

It had been an open problem whether the index data structures such as the suffix array, the CSA, and the FM-index, can be constructed in $o(n \log n)$ time *and* $o(n \log n)$-bit working space. Recently, it was solved by Hon et al. [11].

[⋆] This work was supported by the MOST Grant M6-0203-00-0039.

They proposed an algorithm for constructing the index data structures using $O(n \log\log|\Sigma|)$ time and $O(n\log|\Sigma|)$-bit working space. This algorithm followed the *odd-even scheme* (i.e., $\frac{1}{2}$-recursion), which was used in Farach's algorithm [3, 4] and Kim et al.'s algorithm [13].

In this paper we give another algorithm for constructing the index data structures in $o(n\log n)$ time and $o(n\log n)$-bit working space. Our algorithm constructs the suffix array, the CSA, and FM-index using $O(n)$ time and $O(n\log|\Sigma|\cdot\log^{\alpha}_{|\Sigma|} n)$-bit space, where $\alpha = \log_3 2$. The time complexity of our algorithm is independent of the alphabet size. Hence, our algorithm is the first alphabet-independent linear-time algorithm for constructing the index data structures using $o(n\log n)$-bit space.

The framework of our algorithm follows Kärkkäinen and Sanders's *skew scheme* [12] (i.e., $\frac{2}{3}$-recursion). The merit of our algorithm due to the skew scheme [12] is that the *merging step* is simple compared with Hon et al.'s algorithm [11]. As in Hon et al.'s algorithm [11], moreover, our algorithm don't need the *encoding step*, which is the most complex and time-consuming step in the skew scheme.

It remains an open problem to construct the suffix array, the CSA, and the FM-index *optimally*, i.e., using $O(n)$ time and $O(n\log|\Sigma|)$-bit working space.

2 Preliminaries

In this section we give some basic notations and definitions including the Ψ function of the Compressed Suffix Array (CSA) [7], the array C of the Burrows-Wheeler Transformation (BWT) [1], and the Ψ' function, which is similar to Ψ and the core of our algorithm.

Let T be a string of length n over an alphabet Σ. For simplicity, we assume that n is a multiple of 3. We denote the ith character by $T[i]$ and the substring $T[i]T[i+1]\cdots T[j]$ by $T[i..j]$. We assume that $T[n] = \$$ is a unique terminator which is lexicographically smaller than any other character in Σ. For $1 \le i \le n$, $T[i..n]$ is a *suffix* of T and $T[i..n]T[1..i-1]$ is a *circular string* of T. We denote circular string $T[i..n]T[1..i-1]$ by $T\langle i\rangle$.

For $1 \le i \le n/3$, a suffix $T[3\cdot i-2..n]$, a suffix $T[3\cdot i-1..n]$, and $T[3\cdot i..n]$ are called a *residue-1* suffix, a *residue-2* suffix, and a *residue-3* suffix of T, respectively. Let $T[i..n]$ be lexicographically the kth smallest suffix of T. Then, the *rank* of $T[i..n]$ in the suffixes of T is k. The suffix array of T is a lexicographically sorted array of the suffixes of T. Formally, the suffix array $SA[1..n]$ of T is an array of integers such that $SA[k] = i$, where k is the rank of $T[i..n]$ in the suffixes of T. See Figure 1 for an example.

2.1 Ψ Function and C Array

We define the Ψ_T function [7, 11], or simply Ψ. Let $T[k..n]$ be the suffix stored in the ith entry of SA. Then, $\Psi[i]$ is a position in SA where $T[k+1..n]$ are stored.

i	$C[i]$	$SA[i]$	$\Psi[i]$	
1	a	9	8	$
2	b	8	1	a$
3	b	4	5	aabba$
4	b	2	7	abaabba$
5	a	5	9	abba$
6	b	7	2	ba$
7	a	3	3	baabba$
8	$	1	4	babaabba$
9	a	6	6	bba$

Fig. 1. The suffix array SA, Ψ function, and C array of S = babaabba$.

More formally,

$$\Psi[i] = \begin{cases} SA^{-1}[SA[i]+1] & (\text{if } SA[i] \neq n) \\ SA^{-1}[1] & (\text{if } SA[i] = n) \end{cases}$$

See Figure 1 for an example. The Ψ function is piece-wise increasing. Thus, the Ψ function can be encoded using $O(n \log |\Sigma|)$ bits in the form $T[SA[i]] \times n + \Psi[i] - 1$, which is an increasing sequence, so that each $\Psi[i]$ can be retrieved in constant time [2, 8, 17].

Lemma 1. [11] *Given T and the Ψ function, the suffix array and the compressed suffix array can be constructed in $O(n)$ time and $O(n \log |\Sigma|)$-bit working space.*

The $C[1..n]$ array is defined as

$$C[i] = \begin{cases} T[SA[i]-1] & (\text{if } SA[i] \neq 1) \\ T[n] & (\text{if } SA[i] = 1) \end{cases}$$

See Figure 1 for an example.

Lemma 2. [11] *Given the C array of T, the FM-index can be constructed in $O(n)$ time and $O(n \log |\Sigma|)$-bit working space.*

It is known that Ψ and C are one-to-one corresponding and the transformation between them can be done in linear time using $O(n \log |\Sigma|)$ bits.

Lemma 3. [11] *Given $C[1..n]$, we can compute $\Psi[1..n]$ in $O(n)$ time and $O(n \log |\Sigma|)$ bits.*

Lemma 4. [11] *Given $\Psi[1..n]$ and T, we can construct $C[1..n]$ in $O(n)$ time and $O(n \log |\Sigma|)$ bits.*

2.2 Ψ' Function

Let T_1, T_2 and T_3 be the strings of length $n/3$ over the alphabet Σ^3, which are formed by merging every 3 characters in $T\langle 1\rangle$, $T\langle 2\rangle$ and $T\langle 3\rangle$, respectively.

S_{12} = `bab aab ba$ aba abb a$b`

i	P	C'_{12}	SA_{12}	$\Psi'_{S_{12}}$	F_{12}
1	0	b	6	1	a $b
2	1	b	2	4	a ab ba$
3	0	b	4	2	a ba abb a$b
4	0	a	5	3	a bb a$b
5	1	b	3	1	b a$
6	1	$	1	3	b ab aab ba$

(a) P, SA_{12}, $\Psi'_{S_{12}}$, F_{12} and C'_{12}

S_3 = `baa bba $ba`

i	C'_3	SA_3	Ψ'_{S_3}	F_3
1	a	3	6	$ ba
2	a	1	2	b aa bba $ba
3	a	2	5	b ba $ba

(b) SA_3, Ψ'_{S_3}, F_3 and C'_3

Fig. 2. P array, Ψ' function, F array and C' array for S = `babaabba$`. F and C' are defined in Section 5.

Then, the residue-1, residue-2, and residue-3 suffixes of T correspond one-to-one to the suffixes of T_1, T_2 and T_3, respectively. Note that the last characters of T_1, T_2, and T_3 is unique. We denote the concatenation T_1 and T_2 by T_{12}. Let SA_{12} and SA_3 be the suffix array of T_{12} and T_3, respectively.

Fact 1 *Consider a suffix $T_{12}[i..2n/3]$ $(= T[3i-2..n]T[2..n]T[1])$ for $1 \le i \le n/3$. Because the last character of T_1 is unique, the rank of this suffix is determined by $T[3i-2..n]$. Thus, $T[3i-2..n]$ can be regarded as the suffix stored in the kth entry of SA_{12}, where $k = SA_{12}^{-1}[i]$.*

We divide SA_{12} into two parts. Part 1 and 2 store the suffixes of T_1 and T_2, respectively. Formally, the ith entry of SA_{12} belongs to Part 1 if $1 \le SA_{12}[i] \le n/3$, and it belongs to Part 2 otherwise. The *part array* $P[1..2n/3]$ of SA_{12} is a bit-array representing which part the ith entry of SA_{12} belongs to. We set $P[i] = 1$ if the ith entry of SA_{12} belongs to Part 1, and $P[i] = 0$ otherwise.

Lemma 5. *Given T, $\Psi_{T_{12}}$, and $SA_{12}^{-1}[1]$, we can construct $P[1..2n/3]$ in $O(n)$ time and $O(n)$ bits.*

Proof. For simplicity of notations, we denote $\Psi_{T_{12}}$ by Ψ. Let $t = SA_{12}^{-1}[1]$. For $1 \le i < 2n/3$, $SA_{12}[\Psi^i[t]] = i+1$. By definition, $SA_{12}[\Psi[i]] = SA[i]+1$. Thus, $SA_{12}[\Psi^i[t]] = SA_{12}[\Psi^{i-1}[t]]+1 = \cdots = SA_{12}[\Psi[t]]+i-1 = SA_{12}[t]+i = i+1$.

Hence, we have $P[t] = P[\Psi^i[t]] = 1$ for $1 \le i < n/3$. We set $P[t] = 1$ and initialize $P[j] = 0$ for $j \ne t$. And we iteratively compute $\Psi^i[t]$ and set $P[\Psi^i[t]] = 1$ for $1 \le i < n/3$. The total time required is $O(n)$, and the space is $O(n)$ bits for P.

We define Ψ' of T which plays a central role in our algorithm. Intuitively, the Ψ' function is just like Ψ, but Ψ' is defined in SA_{12} and SA_3. The Ψ' function consists of Part 1 and Part 2 of $\Psi'_{T_{12}}$, and Ψ'_{T_3}. The definition of Ψ' is as follows:

- Part 1 of $\Psi'_{T_{12}}[i]$:
  Let $T[3k-2..n]$ be a suffix stored in the ith entry of SA_{12}, which belongs to Part 1. Then, $\Psi'_{T_{12}}[i]$ is the position in SA_{12} (which belongs to Part 2) where $T[3k-1..n]T[1]$ is stored.

For example, consider $\Psi'_{T_{12}}[2]$ in Figure 2. The suffix stored in the 2nd entry of SA_{12} is $T[4..9]$ (=`aabba$`). $T[5..9]T[1]$ (=`abba$b`) is stored in the 4th entry of SA_{12}. Therefore, $\Psi'_{T_{12}}[2] = 4$.

- Part 2 of $\Psi'_{T_{12}}[i]$:
  Let $T[3k-1..n]T[1]$ be a suffix stored in the ith entry of SA_{12}, which belongs to Part 2. Then, $\Psi'_{T_{12}}[i]$ is the position in SA_3 where $T[3k..n]T[1..2]$ is stored.
  For example, consider $\Psi'_{T_{12}}[4]$ in Figure 2. The suffix stored in the 4th entry of SA_{12} is $T[5..9]T[1]$ (=`abba$b`). $T[6..9]T[1..2]$ (=`bba$ba`) is stored in the 3rd entry of SA_3. Therefore, $\Psi'_{T_{12}}[4] = 3$.
- $\Psi'_{T_3}[i]$:
  Let $T[3k..n]T[1..2]$ be a suffix stored in the ith entry of SA_3. Then, $\Psi'_{T_3}[i]$ is a position in SA_{12} (which belongs to Part 1) where $T[3k+1..n]$ is stored (we assume that when $3k = n$, $T[3k+1..n]$ is $T[1..n]$).
  For example, consider $\Psi'_{T_3}[3]$ in Figure 2. The suffix stored in the 3rd entry of SA_3 is $T[6..9]T[1..2]$ (=`bba$ba`). $T[7..9]$ (=`ba$`) is stored in the 5th entry of SA_{12}. Therefore, $\Psi'_{T_3}[3] = 5$.

More formally,

$$\Psi'_{T_{12}}[i] = \begin{cases} SA_{12}^{-1}[SA_{12}[i] + n/3] \text{ if } 1 \le SA_{12}[i] \le n/3 & \text{(Part 1)} \\ SA_3^{-1}[SA_{12}[i] - n/3] \text{ if } n/3 < SA_{12}[i] \le 2n/3 & \text{(Part 2)} \end{cases}$$

$$\Psi'_{T_3}[i] = \begin{cases} SA_{12}^{-1}[1] & \text{if } SA_3[i] = 3/n \\ SA_{12}^{-1}[SA_3[i]+1] & \text{othersiwe.} \end{cases}$$

Similarly to Ψ, the Ψ' function can be encoded in $O(n \log |\Sigma|)$ bits, so that each $\Psi'_{T_{12}}[i]$ and $\Psi'_{T_3}[i]$ can be retrieved in constant time. Part 1 of $\Psi'_{T_{12}}$ is piece-wise increasing. Thus, we encode Part 1 of the $\Psi'_{T_{12}}$ function in the form $T[3SA_{12}[i]-2] \cdot n + \Psi'_{T_{12}}[i] - 1$, which is an increasing sequence. Similarly, we can encode Part 2 of $\Psi'_{T_{12}}$ and Ψ'_{T_3}.

3 Framework

We will describe how to construct the Ψ function and the C array of T in $O(n)$ time. Then, we can construct the suffix array, the compressed suffix array, and the FM-index in $O(n)$ time and $O(n \log |\Sigma|)$-bit working space using T, the Ψ function, and the C array by Lemmas 1 and 2.

For simplicity, we assume that the length of T is a multiple of $3^{\lceil \log_3 \log_{|\Sigma|} n \rceil + 1}$. Let h be $\lceil \log_3 \log_{|\Sigma|} n \rceil$. We denote $T\langle a_1, \ldots, a_p \rangle$ to be the string formed by concatenating circular strings $T\langle a_1 \rangle, \ldots, T\langle a_p \rangle$ in order. For any string S over Σ, we define $S^{(k)}$ to be the string over the alphabet Σ^{3^k}, which is formed by concatenating every 3^k characters in S to make one character. By definition, $S^{(0)} = S$. For $1 \le k \le h$, we recursively define a string $\mathcal{T}^k$ over Σ^{3^k} as follows. We define $\mathcal{T}^0$ as $T\langle 1 \rangle^{(0)}$ $(= T)$. Let $\mathcal{T}^{k-1} = T\langle a_1, \ldots, a_{2^{k-1}} \rangle^{(k-1)}$. Then,

$$\begin{aligned} \mathcal{T}^k &= T\langle a_1, \ldots, a_{2^{k-1}}, a_{2^{k-1}+1}, \ldots, a_{2^k} \rangle^{(k)} \\ &= T\langle a_1 \rangle^{(k)} \ldots T\langle a_{2^k} \rangle^{(k)}, \end{aligned}$$

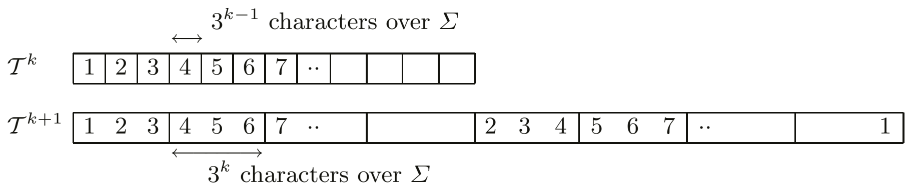

Fig. 3. The relationship between $\mathcal{T}^k$ and $\mathcal{T}^{k+1}$.

where $a_{2^{k-1}+i} = a_i + 3^{k-1}$ for $1 \le i \le 2^{k-1}$. That is, $\mathcal{T}^1 = T\langle 1,2\rangle^{(1)}$, $\mathcal{T}^2 = T\langle 1,2,4,5\rangle^{(2)}$, $\mathcal{T}^3 = T\langle 1,2,4,5,10,11,13,14\rangle^{(3)}$, and so on. The length of $\mathcal{T}^k$ is $(2/3)^k n$.

Fact 2 *For $1 \le i < j \le 2^k$, $a_i < a_j$.*

Lemma 6. *For $1 \le i \le 2^k$, the last character of $T\langle a_i\rangle^{(k)}$ is unique in $\mathcal{T}^k$.*

Proof. We first prove that $a_{2^k} \le 3^k$ by induction. When $k = 0$, $a_{2^0} = 1 = 3^0$. Supposing that $a_{2^{k-1}} \le 3^{k-1}$, $a_{2^k} = a_{2^{k-1}} + 3^{k-1} \le 3^{k-1} + 3^{k-1} < 3^k$.

Because $T\langle a_i\rangle[n - a_i + 1] = \$$ and $a_i \le 3^k$, \$ is contained only in the last character of $T\langle a_i\rangle^{(k)}$. By Observation 2, the position of \$ in $T\langle a_i\rangle$ is different from that in $T\langle a_j\rangle$ for any $j \ne i$. Therefore, the last character of $T\langle a_i\rangle^{(k)}$ is unique in $\mathcal{T}^k$.

Consider the relationship between $\mathcal{T}^k$ and $\mathcal{T}^{k+1}$. See Figure 3. Let $S[1..m]$ be string $\mathcal{T}^k$. Roughly speaking, $\mathcal{T}^{k+1}$ is the string of length $2m/3$, which is formed by merging every 3 characters in $S\langle 1\rangle S\langle 2\rangle$ $(= \mathcal{T}^k\langle 1,2\rangle^{(1)})$. In other words, the suffixes of $\mathcal{T}^{k+1}$ correspond to the residue-1 and residue-2 suffixes of $\mathcal{T}^k$, i.e., $\mathcal{T}^k$'s are essentially the same as the strings made by Kärkkäinen and Sanders's $\frac{2}{3}$-recursion [12].

Lemma 7. *The suffix array of $\mathcal{T}^{k+1}$ is the same as the suffix array of $\mathcal{T}^k\langle 1,2\rangle^{(1)}$.*

Proof. We compare the characters of $\mathcal{T}^{k+1}$ and $\mathcal{T}^k\langle 1,2\rangle^{(1)}$. Let $m = 2^k$ and $X_i[1..p] = T\langle a_i\rangle^{(k)}$, where p must be a multiple of 3. Let X be $X_1[1..p]X_2[1..p] \dots X_m[1..p]$, i.e., $\mathcal{T}^k = X$. Let

$$P = X_1[2..p]\ X_1[1]\ X_2[2..p]\ X_2[1] \dots X_m[2..p]\ X_m[1], \quad \text{and}$$
$$Q = X_1[2..p]\ X_2[1]\ X_2[2..p]\ X_3[1] \dots X_m[2..p]\ X_1[1].$$

Then, $\mathcal{T}^{k+1} = X^{(1)} \cdot P^{(1)}$ and $\mathcal{T}^k\langle 1,2\rangle^{(1)} = X^{(1)} \cdot Q^{(1)}$.

Consider the $(\frac{p}{3} \cdot i)$th characters of $P^{(1)}$ and $Q^{(1)}$, for $1 \le i \le m$.

$$P^{(1)}[p \cdot i/3] = X_i[p-1]\ X_i[p]\ X_i[1] \quad \text{and}$$
$$Q^{(1)}[p \cdot i/3] = X_i[p-1]\ X_i[p]\ X_{i+1}[1].$$

That is, only the 3rd components of these characters are different. However, the 3rd components of these characters don't affect the order of suffixes of $\mathcal{T}^{k+1}$ and $\mathcal{T}^k\langle 1,2\rangle^{(1)}$ because $X_i[p]$ is unique by Lemma 6. The other characters of $\mathcal{T}^{k+1}$ and $\mathcal{T}^k\langle 1,2\rangle^{(1)}$ are the same. Therefore, we get the lemma.

Corollary 1. *The Ψ function of $\mathcal{T}^{k+1}$ is the same as the Ψ function of $\mathcal{T}^k\langle 1,2\rangle^{(1)}$.*

The basic framework to construct Ψ_T $(= \Psi_{\mathcal{T}^0})$ goes bottom-up. That is, we construct $\Psi_{\mathcal{T}^k}$ for $k = h$ down to 0. The algorithm is divided into two phases. Phase 1 consists of step h and Phase 2 consists of the remaining h steps. The details of each phase are as follows.

For Phase 1, we construct $\Psi_{\mathcal{T}^h}$ by first building the suffix array of $\mathcal{T}^h$ using any linear-time construction algorithm [4, 12–14], and then converting it to $\Psi_{\mathcal{T}^h}$.

In step k of Phase 2, we construct $\Psi_{\mathcal{T}^k}$ from $\Psi_{\mathcal{T}^{k+1}}$. Recall that $\mathcal{T}^k\langle 1,2\rangle^{(1)}$ and $\mathcal{T}^k\langle 3\rangle^{(1)}$ are denoted by $\mathcal{T}^k_{12}$ and $\mathcal{T}^k_3$, respectively. We first compute $\Psi'_{\mathcal{T}^k_{12}}$ and $\Psi'_{\mathcal{T}^k_3}$ using $\mathcal{T}^k$ and $\Psi_{\mathcal{T}^{k+1}}$. Then, we construct $\Psi_{\mathcal{T}^k}$ by merging $\Psi'_{\mathcal{T}^k_{12}}$ and $\Psi'_{\mathcal{T}^k_3}$.

In Sections 4 we describe how to compute $\Psi'_{\mathcal{T}^k_{12}}$ and $\Psi'_{\mathcal{T}^k_3}$ from $\mathcal{T}^k$ and $\Psi_{\mathcal{T}^{k+1}}$ $(= \Psi_{\mathcal{T}^k_{12}}$ by Corollary 1) in $O(|\mathcal{T}^k|+|\Delta|)$ time and $O(|\mathcal{T}^k|\log|\Delta|+|\Delta|)$-bit space, where Δ is the alphabet of $\mathcal{T}^k$. In Sections 5 we describe how to merge $\Psi'_{\mathcal{T}^k_{12}}$ and $\Psi'_{\mathcal{T}^k_3}$ in $O(|\mathcal{T}^k|)$ time and $O(|\mathcal{T}^k|\log|\Delta|)$-bit space.

Theorem 1. *The Ψ function of T can be constructed in $O(n)$ time and $O(n\log|\Sigma|\cdot\log^{\alpha}_{|\Sigma|} n)$-bit space, where $\alpha = \log_3 2$.*

Proof. For Phase 1, we first construct the suffix array for $\mathcal{T}^h$ whose size is $n(2/3)^{\log_3\log_{|\Sigma|} n} \le n(\log_{|\Sigma|} n)^{\alpha-1}$. This requires $O(n)$ time and $O(n\log|\Sigma|\cdot\log^{\alpha}_{|\Sigma|} n)$-bit space by using any linear-time construction algorithm [4, 12–14]. Then $\Psi_{\mathcal{T}^h}$ can be constructed in $O(n)$ time and $O(n\log|\Sigma|\cdot\log^{\alpha}_{|\Sigma|} n)$-bit space. Thus, Phase 1 takes $O(n)$ time and $O(n\log|\Sigma|\cdot\log^{\alpha}_{|\Sigma|} n)$-bit space.

For every step i in Phase 2, we construct $\Psi_{\mathcal{T}^i}$. Let Δ_i be the alphabet of $\mathcal{T}^i$. For the space, each step requires $O(|\mathcal{T}^i|\log|\Delta_i| + |\Delta_i|)$ bits. Note that $|\mathcal{T}^i| = (2/3)^i n$ and $|\Delta_i| \le |\Sigma|^{3^i} \le n$, so $|\mathcal{T}^i|\log|\Delta_i| = (2/3)^i n\log|\Sigma|^{3^i} = 2^i n\log|\Sigma| \le n\log|\Sigma|\cdot\log^{\alpha}_{|\Sigma|} n$. Therefore, the space for Phase 2 is $O(n\log|\Sigma|\cdot\log^{\alpha}_{|\Sigma|} n)$ bits. The time of each step is $O((|\mathcal{T}^i| + |\Delta_i|)$. The total time of Phase 2 is

$$\sum_{i=1}^{\lceil\log_3\log_{|\Sigma|} n\rceil-1} O\left(n\left(\frac{2}{3}\right)^i + |\Sigma|^{3^i}\right) = O(n).$$

Finally, consider the space storing $\mathcal{T}^i = T\langle a_1,\ldots,a_{2^i}\rangle^{(i)}$. We don't store $\mathcal{T}^i$ explicitly but the values of a_j $(1 \le j \le 2^i)$. We can get a character of $\mathcal{T}^i$ in constant time using the values of a_j and T. The circular strings of T which compose $\mathcal{T}^h$ include all those which compose $\mathcal{T}^i$. Therefore, we just store 2^h integers of size $\log n$ for the whole algorithm. The space is $\log n\cdot\log^{\alpha}_{|\Sigma|} n$ bits.

i	P	X_i		
		$x[i]$	$S_{12}[SA_{12}[i]..m/3]$	$S[1..2]$
1	0	·	·	·
2	1	b	aa bba \$	ba
3	0	·	·	·
4	0	·	·	·
5	1	b	ba \$	ba
6	1	\$		ba

(a) $x[i]$ and X_i

k	sorted x	$i \rightarrow \Psi'_{S_3}$
1	\$	6
2	b	2
3	b	5

(b) $x[i]$ and index i after stable sorting on x

Fig. 4. Consider S = babaabba\$. For comparison, see Figure 2.

4 Constructing Ψ'

Let $S[1..m]$ be $\mathcal{T}^k$. We define S_{12} and S_3 just as $\mathcal{T}^k_{12}$ and $\mathcal{T}^k_3$, respectively. We denote the suffix arrays of S_{12} and S_3 by SA_{12} and SA_3, respectively. Let Δ be the alphabet of S, and P be the part array of SA_{12}. We assume $S[0] = S[m]$.

Given $S[1..m]$, $\Psi_{S_{12}}$, and $SA^{-1}_{12}[1]$, we will describe how to construct $\Psi'_{S_{12}}$ and Ψ'_{S_3}. The algorithm consists of three parts: Constructing Ψ'_{S_3} using Part 1 of $\Psi_{S_{12}}$, constructing Part 2 of $\Psi'_{S_{12}}$ using Ψ'_{S_3}, and constructing Part 1 of $\Psi'_{S_{12}}$ using Part 2 of $\Psi_{S_{12}}$. We describe only how to construct Ψ'_{S_3} using Part 1 of $\Psi_{S_{12}}$. Part 1 and 2 of $\Psi'_{S_{12}}$ can be constructed similarly.

We define $x[1..2m/3]$ as an array of characters such that $x[i] = S[3SA_{12}[i]-3]$ if $P[i] = 1$, and $x[i]$ isn't defined otherwise. For i with $P[i] = 1$, let X_i be the string $x[i]S_{12}[SA_{12}[i]..m/3]S[1..2]$ if $SA_{12}[i] \neq 1$, and the string $x[i]S[1..2]$ $(= S[n]S[1..2])$ otherwise. For i with $P[i] = 0$, X_i isn't defined. See Figure 4 (a) for an example. From now on, we consider only $x[i]$'s and X_i's such that $P[i] = 1$. Let X be the set $\{X_i \mid P[i] = 1, 1 \leq i \leq 2m/3\}$. Then, X_i is a suffix of S_3 and X is the same as the set of suffixes of S_3.

Lemma 8. *The stable sorting order of $x[i]$ is equal to the rank of X_i in X.*

Proof. Let X_p be the element of X such that $SA_{12}[i] = 1$. By omitting the first characters of every X_k's except X_p, they are of the form $S_{12}[SA_{12}[i]..m/3]S[1..2]$, which are already sorted in SA_{12} (note that $S[1..2]$ does not affect the order because $S_{12}[m/3]$ is unique). The first character of X_p $(= S[m])$ is unique. Thus, the rank of X_i is equal to the stable sorting order of $x[i]$.

Lemma 9. *Let k be the stable sorting order of $x[i]$ with $P[i] = 1$. Then $\Psi'_{S_3}[k] = i$.*

Proof. Let $S[p..n]$ be the suffix stored in the ith entry of SA_{12}. Then, $x[i]$ is $S[p-1]$ and the rank of X_i $(= S[p-1..n]S[1..2])$ is k in X by Lemma 8. By definition, $\Psi'_{S_3}[k]$ is a position in SA_{12} where $S[p..n]$ is stored. Therefore, $\Psi'_{S_3}[k] = i$. See Figure 4 (b).

Lemma 10. *Given S, $\Psi_{S_{12}}$, and $SA^{-1}_{12}[1]$, Ψ'_{S_3} can be constructed in $O(m+|\Delta|)$ time and $O(m \log |\Delta| + |\Delta|)$-bit space.*

```
Procedure Const_C_array
begin
  $i \leftarrow 1$; $j \leftarrow 1$;
  for $k = 1$ to $m$
    $rval \leftarrow$ Compare_suffix$(i, j)$;
    if $rval > 0$ then
      $C[k] \leftarrow C'_{12}[i]$;
      $i \leftarrow i + 1$;
    else
      $C[k] \leftarrow C'_3[j]$;
      $j \leftarrow j + 1$;
end
```

```
Function Compare_suffix$(i, j)$
begin
  if $P[i] = 1$ then
    $x \leftarrow (F_{12}[i], \Psi'_{S_{12}}[i])$;
    $y \leftarrow (F_3[j], \Psi'_{S_3}[j])$;
  else
    $x \leftarrow (F_{12}[i], F_3[\Psi'_{S_{12}}[i]], \Psi'_{S_3}[\Psi'_{S_{12}}[i]])$;
    $y \leftarrow (F_3[j], F_{12}[\Psi'_{S_3}[j]], \Psi'_{S_{12}}[\Psi'_{S_3}[j]])$;
  if $x < y$ then
    return 1;
  else return -1;
end
```

Fig. 5. Constructing the C array of S.

Proof. Given S, $\Psi_{S_{12}}$, and $SA_{12}^{-1}[1]$, we first construct x in $O(m)$ time and $O(m \log |\Delta|)$ bits using a method similar to that in Lemma 5.

Then, for $i = 1$ to $2m/3$ with $P[i] = 1$, we iteratively compute the stable sorting order k of $x[i]$ and set $\Psi'_{S_3}[k] = i$. The total iteration of the stable sorting can be performed in $O(m + |\Delta|)$ time using $O(m \log |\Delta| + |\Delta|)$ bits [11].

Similarly, we can construct Part 2 of $\Psi'_{S_{12}}$ using Ψ'_{S_3} and Part 1 of $\Psi'_{S_{12}}$ using Part 2 of $\Psi_{S_{12}}$. Thus, we get the following lemma.

Lemma 11. *Given S, $\Psi_{S_{12}}$, and $SA_{12}^{-1}[1]$, we can construct $\Psi'_{S_{12}}$ and Ψ'_{S_3} in $O(m + |\Delta|)$ time and $O(m \log |\Delta| + |\Delta|)$-bit space.*

5 Merging $\Psi'_{S_{12}}$ and Ψ'_{S_3}

In this section we will describe how to construct Ψ_S by merging $\Psi'_{S_{12}}$ and Ψ'_{S_3}. We first construct the C array of S by merging $\Psi'_{S_{12}}$ and Ψ'_{S_3}. Merging $\Psi'_{S_{12}}$ and Ψ'_{S_3} is similar to Kärkkäinen and Sanders's algorithm [12], which merge SA_{12} and SA_3 in $O(m)$ time. Then, we convert C to Ψ_S by Lemma 3.

Let $F_{12}[1..2m/3]$ and $F_3[1..m/3]$ be arrays of the first characters, over Δ, of the suffixes in SA_{12} and S_3, respectively. That is, $F_{12}[i] = S[3SA_{12}[i] - 2]$ if $P[i] = 1$, and $F_{12}[i] = S[3(SA_{12}[i] - m/3) - 1]$ otherwise, and $F_3[i] = S[3SA_3[i]]$. Similarly, let $C'_{12}[1..2m/3]$ and $C'_3[1..m/3]$ be arrays of the characters preceding F_{12} and F_3 in S, respectively. That is, $C'_{12}[i] = S[3SA_{12}[i] - 3]$ if $P[i] = 1$, and $C'_{12}[i] = S[3(SA_{12}[i] - m/3) - 2]$ otherwise, and $C'_3[i] = S[3SA_3[i] - 1]$. Note that characters in C'_{12} and C'_3 compose C. See Figure 2 for an example. We can construct these arrays in $O(m)$ time and $O(m \log |\Delta|)$ bits as in Lemma 5.

We construct the C array by merging SA_{12} and SA_3. This merging is similar to merging two sorted arrays of integers. Figure 5 shows Procedure Const_C_array, which constructs the C array. Procedure Const_C_array consists of m iterations. Let $S[p..m]$ be the suffix in the ith entry of SA_{12} and $S[q..m]$ be the suffix in the jth entry of SA_3 (note that we ignore characters following $S[m]$

in SA_{12} and S_3 because these characters do not affect the order of suffixes). In the kth iteration, we determine which suffix is lexicographically the kth smallest by comparing $S[p..m]$ with $S[q..m]$, and thus we can compute $C[k]$. During the merging stage, we can get $SA^{-1}[1]$ which will be used in the next step.

Function `Compare_suffix`(i, j) compares $S[p..m]$ and $S[q..m]$ using the Ψ' function and arrays F_{12} and F_3. We have two cases according to the values of $P[i]$. Consider the case of $P[i] = 1$. Then, $S[p..m]$ is a residue-1 suffix and $S[q..m]$ is a residue-3 suffix. We first compare $S[p]$ $(= F_{12}[i])$ with $S[q]$ $(= F_3[j])$. If $S[p] = S[q]$, we compare $S[p+1..m]$ with $S[q+1..m]$. Because $S[p+1..m]$ and $S[q+1..m]$ are residue-2 and residue-1 suffixes, respectively, $\Psi'_{S_{12}}[i]$ and $\Psi'_{S_3}[j]$ represent the ranks of $S[p+1..m]$ and $S[q+1..m]$ in SA_{12}, respectively. Therefore, we can determine which suffix is smaller by comparing one pair of characters and one pair of integers. Similarly, we can do by comparing two pairs of characters and one pair of integers in case of $P[i] = 0$.

Function `Compare_suffix`(i, j) takes constant time and so Procedure `Const_C_array` takes $O(m)$ time. The space for arrays and Ψ' function is $O(m \log |\Delta|)$ bits. By Lemma 3, we convert C to Ψ_S in $O(m)$ time and $O(m \log |\Delta|)$ bits. Hence, we get the following lemma.

Lemma 12. *Given S, $\Psi'_{S_{12}}$, Ψ'_{S_3}, and $SA_{12}^{-1}[1]$, we can compute Ψ_S and $SA^{-1}[1]$ in $O(m)$ time and $O(m \log |\Delta|)$-bit space.*

References

1. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Paolo Alto, California, 1994.
2. D. R. Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, Waterloo, 1988.
3. M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science*, pages 137–143, 1997.
4. M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000.
5. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science*, pages 390–398, 2000.
6. G. H. Gonnet, R. Baeza-Yates, and T. Snider. New indices for text: Pat trees and pat arrays. In W. B Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*, pages 66–82. Prentice Hall, 1992.
7. R. Grossi and J. S. Vitter. 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*, pages 397–406, 2000.
8. R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. Technical Report Submitted for publication, 2001.

9. D. Gusfield. An “Increment-by-one” approach to suffix arrays and trees. manuscript, 1990.
10. W. K. Hon, T. W. Lam, K. Sadakane, and W. K. Sung. Constructing compressed suffix arrays with large alphabets. In *Proceedings of the 14th International Symposium on Algorithms and Computation*, pages 240–249, 2003.
11. W. K. Hon, K. Sadakane, and W. K. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, pages 251–260, 2003.
12. J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proceedings of the 30th International Colloquium on Automata, Languages, and Programming*, pages 943–955, 2003.
13. D.K. Kim, J. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proceedings of the 14th Symposium on Combinatorial Pattern Matching*, pages 186–199, 2003.
14. P. Ko and S. Aluru. Space-efficient linear time construction of suffix arrays. In *Proceedings of the 14th Symposium on Combinatorial Pattern Matching*, pages 200–210, 2003.
15. T. W. Lam, K. Sadakane, W. K. Sung, and S. M. Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. In *Proceedings of the 9th International Computing and Combinatorics Conference*, pages 401–410, 2002.
16. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
17. J. I. Munro. Tables. In *Proceedings of Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 37–42, 1996.