Constructing Compressed Suffix Arrays with Large Alphabets *

Wing-Kai Hon[1], Tak-Wah Lam[1], Kunihiko Sadakane[2], and Wing-Kin Sung[3]

[1] Department of Computer Science and Informations Systems,
The University of Hong Kong, Hong Kong,
{wkhon,twlam}@csis.hku.hk

[2] Department of Computer Science and Communication Engineering,
Kyushu University, Japan,
sada@csce.kyushu-u.ac.jp

[3] School of Computing, National University of Singapore, Singapore,
ksung@comp.nus.edu.sg

Abstract. Recent research in compressing suffix arrays has resulted in two breakthrough indexing data structures, namely, compressed suffix arrays (CSA) [7] and FM-index [5]. Either of them makes it feasible to store a full-text index in the main memory even for a piece of text data with a few billion characters (such as human DNA). However, constructing such indexing data structures with limited working memory (i.e., without constructing suffix arrays) is not a trivial task. This paper addresses this problem. Currently, only CSA admits a space-efficient construction algorithm [15]. For a text T of length n over an alphabet Σ, this algorithm requires $O(|\Sigma| n \log n)$ time and $(2H_0 + 1 + \epsilon)n$ bits of working space, where H_0 is the 0-th order empirical entropy of T and ϵ is any non-zero constant. This algorithm is good enough when the alphabet size $|\Sigma|$ is small. It is not practical for text data containing protein, Chinese or Japanese, where the alphabet may include up to a few thousand characters.

The main contribution of this paper is a new algorithm which can construct CSA in $O(n \log n)$ time using $(H_0 + 2 + \epsilon)n$ bits of working space. Note that the running time of our algorithm is independent of the alphabet size and the space requirement is smaller as it is likely that $H_0 > 1$. This paper also makes contribution to the space-efficient construction of FM-index. We show that FM-index can indeed be constructed from CSA directly in $O(n)$ time.

1 Introduction

The advance in information technology and bio-technology has generated an enormous amount of text data. In particular, a lot of such texts have no word

* This work was supported in part by the Hong Kong RGC Grant HKU-7024/01E; by the Grant-in-Aid of the Ministry of Education, Science, Sports and Culture of Japan; and by the NUS Academic Research Grant R-252-000-119-112.

boundary. Typical examples include DNA, protein, Chinese, and Japanese. A simple piece of such texts can contain millions or even billions of characters. To assist users to locate their required information efficiently, it is vital to index the text using some data structures so as to exploit fast searching algorithms. For text with word boundary (e.g., English), inverted index [9] can be used; this data structure enables fast query and is space efficient. But inverted index cannot be used for text without word boundary. In this case, suffix tree [19] and suffix array [17] are the most useful data structures. They have applications in numerous areas including digital library [20], text data mining [24] and biological research [10].

Suffix tree is a powerful index because it allows us to search pattern using time linear to the pattern length, independent of the text size. For a sequence of n characters over an alphabet Σ, building a suffix tree takes $O(n)$ time. A pattern P can then be located in $O(|P| + occ)$ time, where occ is the number of occurrences. For suffix arrays, construction and searching takes time $O(n)$ [3, 13] and $O(|P| \log n + occ)$, respectively. However, the suffix tree and the suffix array demands a lot of memory space. Both data structures require $O(n \log n)$ bits; the constant associated with suffix arrays is smaller, though. For human DNA (of length approximately 3 billion), the best known implementation of suffix tree and suffix array requires 40 Gigabytes and 13 Gigabytes, respectively [14]. For text data acquired from several years of a Chinese news source[4] (of length approximately 1.3 billion) [6], suffix tree and suffix array require 18 Gigabytes and 8 Gigabytes, respectively. Such memory requirement far exceeds the capacity of ordinary computers (PCs nowadays have up to 4 Gigabytes of main memory). To solve the memory space problem, it has been proposed to store the indexing data structure in the secondary storage [2, 12]. But the searching time deteriorates a lot.

Recently, Grossi and Vitter [7] have proposed a space-efficient indexing data structure called compressed suffix array (CSA), which reduces the space requirement from $O(n \log n)$ bits to $O(n \log |\Sigma|)$ bits, and more importantly, the compressed suffix array still supports searching efficiently. Precisely, the searching time increases by a factor of at most $\log n$. In practice, a CSA for the human DNA occupies around 2 Gigabytes, and it is possible to store the CSA in the main memory of a PC.

To construct a compressed suffix array, a naive way is to build the suffix array first and then convert it to the compressed suffix array. This construction method takes $O(n \log n)$ time, but it requires much more than $O(n \log |\Sigma|)$-bit working space. In other words, it is feasible to store a CSA for human DNA in the main memory of a PC, but it is not feasible to build it on a PC. This definitely limits the application of CSA. To solve this working memory problem, Lam et al. [15] initiates the study of constructing the compressed suffix array directly. In particular, they gave an algorithm that uses only $O(n \log |\Sigma|)$ bits of memory and runs in $O(|\Sigma| n \log n)$ time. This algorithm can index the whole human DNA on a PC in 20 hours. The idea of [15] is to build the compressed

[4] The news source is the Central News Agency of Taiwan.

suffix array incrementally, character by character. With the help of a modified red-black tree, for every insertion of a character, the compressed suffix array can be updated in $O(|\Sigma| \log n)$ time. Since the text has n characters, the compressed suffix array can be obtained using $O(|\Sigma| n \log n)$ time.

For DNA (which has only four distinct characters) or texts of very small alphabet, the efficiency of the above algorithm is still acceptable. But for indexing other kinds of texts such as Chinese or Japanese, whose alphabet consists of at least a few thousand characters, the efficiency of the above algorithm deteriorates quickly. This paper gives a more practical algorithm for constructing compressed suffix arrays, reducing the time from $O(|\Sigma| n \log n)$ to $O(n \log n)$, while maintaining the $O(n \log |\Sigma|)$-bit space complexity.[5] In fact, our algorithm demands less working space; more precisely, it reduces the $(2H_0 + 1 + \epsilon)n$ space requirement in [15] to $(H_0 + 2 + \epsilon)n$, where ϵ is any non-zero constant, and H_0 is the 0-th order empirical entropy of the text. Note that H_0 is at most $\log |\Sigma|$ and is very likely to be bigger than 1. Thus, our algorithm allows us to index even longer text than the previous algorithm on ordinary PCs.

Instead of building a CSA character by character, the new algorithm partitions the text into $\log n$ segments where each segment is of length $n/\log n$. Suppose we have already built a CSA for the first i segments, we show that we can "merge" the $(i+1)$-th segment with the CSA to form a bigger CSA in $O(n)$ time. Thus, the total time to build the whole compressed suffix array is $O(n \log n)$. Technically speaking, the new algorithm avoids using a balance tree of short sequences as an intermediate representation. The updating of such short sequences is inherently slow. Instead, we devise a more elegant representation based on Rice code [21, 8].

In addition to CSA, Ferragina and Manzini [5] have proposed another scheme called FM-index to compress suffix arrays. The FM-index can match the storage requirement and searching efficiency of CSA, and it may be even better in some cases. Yet there is also no direct way to construct FM-index in the literature. Another contribution of this paper is the first algorithm for constructing FM-index using limited working space. We show that once CSA is constructed, we can build FM-index from CSA in $O(n)$ time without extra working storage.

The rest of this paper is organized as follows. Section 2 gives a review of suffix arrays and compressed suffix arrays. Section 3 presents the new construction algorithm of CSA, while Section 4 discusses the space-efficient construction of FM-index. We conclude the paper in Section 5.

2 Preliminaries

Let Σ be an alphabet, and \$ be a special character not in Σ. We assume that \$ is lexicographically smaller than any character in Σ, which is used to mark the end of a text. Consider a length-n text T defined on Σ, which is represented

[5] The quest for construction algorithm of text indices tailor-made for large alphabets is not a new idea. For instance, Farach [3] extends the linear-time suffix tree construction algorithm from constant-sized alphabet to general alphabets.

by an array $T[0..n] = T[0]T[1]\ldots T[n]$, where $T[n] = \$$. For $i = 0, 1, 2, \ldots, n$, $T_i = T[i..n] = T[i]T[i+1]\ldots T[n]$ denotes a suffix of T starting from the i-th position.

i	$T[i]$	T_i
0	a	$acaaccg\$$
1	g	$caaccg\$$
2	a	$aaccg\$$
3	a	$accg\$$
4	g	$ccg\$$
5	g	$cg\$$
6	c	$g\$$
7	\$	\$

i	$\mathrm{SA}[i]$	$T_{\mathrm{SA}[i]}$
0	7	\$
1	2	$aaccg\$$
2	0	$acaaccg\$$
3	3	$accg\$$
4	1	$caaccg\$$
5	4	$ccg\$$
6	5	$cg\$$
7	6	$g\$$

i	$\Psi[i]$	$T[\mathrm{SA}[i]]$
0	2	\$
1	3	a
2	4	a
3	5	a
4	1	c
5	6	c
6	7	c
7	0	g

Fig. 1. Suffixes, suffix array, and compressed suffix array of $acaaccg\$$

A suffix array [17] is a sorted sequence of $n+1$ suffixes of T, denoted by $\mathrm{SA}[0..n]$. Formally, $\mathrm{SA}[0..n]$ is a permutation of the set of integers $\{0, 1, \ldots, n\}$ such that, according to the lexicographic order, $T_{\mathrm{SA}[0]} < T_{\mathrm{SA}[1]} < \ldots < T_{\mathrm{SA}[n]}$. See Figure 1 for an example. That is, $\mathrm{SA}[i]$ denotes the starting position of the i-th smallest suffix of T. Note that $\mathrm{SA}[0] = n$, so that for the remaining SA values, each can be represented in $\log n$ bits, and the suffix array can thus be stored using $(n+1)\log n$ bits.[6] Given a text T together with the suffix array $\mathrm{SA}[0..n]$, the occurrences of any pattern P in T can be found without scanning T again. Precisely, it takes $O(|P|\log n + occ)$ time, where occ is the number of occurrences.

For every $i = 0, 1, 2, \ldots, n$, define $\mathrm{SA}^{-1}[i]$ to be the integer j such that $\mathrm{SA}[j] = i$. Intuitively, $\mathrm{SA}^{-1}[i]$ denotes the *lex-order* of T_i among the suffixes of T, which is the number of suffixes of T lexicographically smaller than or equal to the suffix T_i.

In general, to express the relation of a string X among the suffixes of T, we use $order(X, T)$ to denote the lex-order of X among all the suffixes of T. Thus, $\mathrm{SA}^{-1}[i] = order(T_i, T)$.

Based on SA and SA^{-1}, the compressed suffix array [7, 22] for a text T stores an array $\Psi[0..n]$ where $\Psi[i] = \mathrm{SA}^{-1}[\mathrm{SA}[i]+1]$ for $i = 1, 2, \ldots, n$, while $\Psi[0]$ is defined as $\mathrm{SA}^{-1}[0]$. See Figure 1 for an example.

Note that $\Psi[0..n]$ contains $n+1$ integers. A trivial way to store the function requires $(n+1)\log n$ bits, which is the same space as storing the SA. Nevertheless, $\Psi[1..n]$ can always be partitioned into $|\Sigma|$ strictly increasing sequences, which allows it to be stored succinctly. This increasing property is illustrated in the rightmost table in Figure 1 and the correctness is proved based on the following lemmas.

[6] Throughout this paper, we assume that the base of the logarithm is 2.

Lemma 1. ([15]) *For every $i < j$, if $T[\mathrm{SA}[i]] = T[\mathrm{SA}[j]]$, then $\Psi[i] < \Psi[j]$.*

For any character $c \in \Sigma$, define l_c to be the maximum p such that $T_{\mathrm{SA}[p-1]} < c$ and r_c to be the maximum p such that $T_{\mathrm{SA}[p]} \leq c$.

Lemma 2. ([15]) *If $l_c \leq r_c$, $\Psi[l_c \ldots r_c]$ is strictly increasing.*

Corollary 1. *$\Psi[1..n]$ can be partitioned into at most $|\Sigma|$ strictly increasing sequences.*

Based on the increasing property, Grossi and Vitter [8] showed that Ψ can be encoded in $(H_0 + 2 + o(1))n$ bits using Rice code [21]. Moreover, each Ψ value can be retrieved in $O(1)$ time. Based on their encoding scheme, we observe a further property which is stated as follows.

Lemma 3. *Suppose that we can enumerate $\Psi[i]$ sequentially for $i = 1, 2, \ldots, n$. Then in $O(n)$ time, we can encode the Ψ function using $(H_0 + 2 + o(1))n$ bits.*

3 Our Algorithm

We want to construct $\Psi[1 \ldots n]$ for the text T. To do so, we first partiton the text into n/ℓ consecutive regions, say $T^1, T^2, \ldots, T^{n/\ell}$, each with $\ell = O(\frac{n}{\log n})$ characters.[7] The algorithm builds the Ψ function incrementally, starting with that of $T^{n/\ell}$, and then $T^{n/\ell-1}T^{n/\ell}, \ldots$ and finally obtains the required Ψ function of $T^1T^2 \ldots T^{n/\ell} = T$. There are two main steps.

1. Base Step: Compute the Ψ function for $T^{n/\ell}$.
2. Merge Step: For $i = n/\ell - 1, n/\ell - 2, \ldots, 1$, compute the Ψ function of T^iT' based on T^i and the Ψ function of T', where T' denotes the string $T^{i+1}T^{i+2} \ldots T^{n/\ell}$.

3.1 Time Complexity: Base Step

To compute the Ψ function of $T^{n/\ell}$, we first compute the SA for $T^{n/\ell}$ by suffix sorting. Since the length of $T^{n/\ell}$ is ℓ, suffix sorting takes $O(\ell \log \ell)$ time [16]. Afterwards, the SA^{-1} function can be computed in $O(\ell)$ time. Then together with the SA function, we obtain the Ψ function in $O(\ell)$ time. The overall time for the Base Step is therefore $O(\ell \log \ell)$.

3.2 Time Complexity: Merge Step

Consider $i \in \{1, 2, \ldots, n/\ell - 1\}$, and let $T^i = c_1c_2 \cdots c_\ell$. Let $suf_1, suf_2, \ldots, suf_\ell$ to be the ℓ longest suffixes of T^iT'. Precisely, $suf_k = c_kc_{k+1} \cdots c_\ell T'$ for $k = 1, \ldots, \ell$.

The Iterative Step can be sub-divided into three parts:

[7] That is, we let $T^i = T[(i-1)\ell]\, T[(i-1)\ell + 1] \ldots T[i\ell - 1]$.

(a) Sort the suffixes $suf_1, suf_2, \ldots, suf_\ell$ to find the lex-orders of each suffix among themselves.
(b) For every suf_i, calculate $order(suf_i, T')$.
(c) Compute the Ψ function for T^iT'.

Part (a) can be done based on the following lemma.

Lemma 4. *The ℓ longest suffixes of T^iT' can be sorted in $O(\ell \log \ell)$ time.*

Let $\mathrm{SA}_{T'}$ denote the suffix array of T', $\mathrm{SA}_{T'}^{-1}$ denote the inverse of the $\mathrm{SA}_{T'}$, and $\Psi_{T'}$ denote the corresponding Ψ function of T'. For any character c, we use the notation of l_c and r_c to denote the maximum p such that $T'_{\mathrm{SA}_{T'}[p-1]} < c$ and the maximum p such that $T'_{\mathrm{SA}_{T'}[p]} \leq c$, respectively. The lemmas below show how to compute $order(c_k c_{k+1} \cdots c_\ell T', T')$ iteratively for $k = \ell, \ell - 1, \ldots, 1$, thus achieving Part (b).

Lemma 5. *Let X be any string and c be any character. Let B denote the set $\{b \mid b \in [l_c, r_c] \wedge \Psi_{T'}[b] \leq order(X, T')\}$. Then,*

$$order(cX, T') = \begin{cases} l_c - 1 & \textit{if } B \textit{ is empty} \\ \max\{b \mid b \in B\} & \textit{otherwise} \end{cases}$$

Lemma 6. *Given the Ψ function of T'. Computing $order(c_k c_{k+1} \cdots c_l T', T')$ for all $1 \leq k \leq \ell$ can be done in $O(\ell \log n)$ time.*

We next discuss Part (c), which computes the Ψ function for T^iT'. Note that by definition, a Ψ function is basically an array of lex-orders of the suffixes of T^iT' among themselves, enumerated in some specific order. To compute Ψ for T^iT', we have to know $order(s, T^iT')$ for every suffix s of T'. We also need $order(suf_k, T^iT')$ for every $1 \leq k \leq l$.

In the following, we present a function f which maps the lex-orders for the existing suffixes of T' in T' to those in T^iT', and a function g which maps the lex-orders for the newly added suffixes (i.e., the ℓ longest suffixes) to those in T^iT'. These two functions compute all the required lex-orders, and are then used to construct the required Ψ function of T^iT'.

Firstly, the following lemma shows the function f that relates $\mathrm{SA}_{T'}^{-1}[m]$ and $\mathrm{SA}_{T^iT'}^{-1}[m']$, where m and m' are positions in T' and T^iT' that correspond to the same suffix of T' (i.e., $m' = m + |T^i|$).

Recall that suf_k denotes the string $c_k c_{k+1} \cdots c_\ell T'$.

Lemma 7. *Suppose $\mathrm{SA}_{T'}^{-1}[m] = j$. Then $\mathrm{SA}_{T^iT'}^{-1}[m']$ is equal to*

$$f(j) = j + \#(order(suf_k, T') \leq j),$$

where $\#(order(suf_k, T') \leq j)$ denotes the number of $order(suf_k, T')$ that is smaller than or equal to j, for $1 \leq k \leq \ell$.

Observation 1 *f is strictly increasing for $j \in [1, |T'|]$.*

Lemma 8. *f can be stored in $o(n)$ bits in $O(n)$ time, so that each $f(j)$ can be computed in constant time.*

The function g, which calculates the lex-orders of the remaining suffixes of T^iT', is shown in the next lemma.

Lemma 9. *For all $j \in [1, \ell]$, $\mathrm{SA}^{-1}_{T^iT'}[j]$ is equal to*

$$g(j) = order(suf_j, T') + \#(suf_k \leq suf_j),$$

where $\#(suf_k \leq suf_j)$ denotes the number of suf_k that is smaller than or equal to suf_j, for all $1 \leq k \leq \ell$.

Observation 2 *For every j, k, if $suf_j < suf_k$, then $g(j) < g(k)$. In other words, $g(k)$ is strictly increasing in the ascending lex-order of suf_k.*

Lemma 10. *The values of the strictly increasing sequences f and g in Observations 1 and 2 are distinct and they span $[1, |T^iT'|]$.*

Based on Lemmas 7, 9 and 10, we are ready to describe how to accomplish Part (c). Let $\Phi = \Psi_{T'}$ and $\Phi' = \Psi_{T^iT'}$.

Lemma 11.
1. $\Phi'[f(j)] = f(\Phi[j])$ *for* $j = 1, 2, \ldots, |T'|$.
2. $\Phi'[g(\ell)] = f(\Phi[0])$.
3. $\Phi'[g(j)] = g(j+1)$ *for* $j = 1, 2, \ldots, \ell - 1$.

See Figures 2 and 3 for an example of the function Φ and Φ', and their relations with f and g.

i	$\mathrm{SA}_{T'}[i]$	$\Phi[i]$	$T'_{\mathrm{SA}_{T'}[i]}$
0	5	4	\$
1	3	3	*ac*\$
2	1	5	*agac*\$
3	4	0	*c*\$
4	0	2	*cagac*\$
5	2	1	*gac*\$

i	$\mathrm{SA}_{T^1T'}[i]$	$\Phi'[i]$	$T_{\mathrm{SA}_{T^1T'}[i]}$
0	8	8	\$
1	6	4	*ac*\$
2	2	6	*acagac*\$
3	4	7	*agac*\$
4	7	0	*c*\$
5	1	2	*cacagac*\$
6	3	3	*cagac*\$
7	5	1	*gac*\$
8	0	5	*gcacagac*\$

Fig. 2. Functions Φ and Φ'. Φ and Φ' denote the Ψ functions for the text $T' = cagac\$$ and the text $T^1T' = gcacagac\$$, respectively.

The algorithm to compute the required Φ' is as follows. By Lemma 10, merging the increasing sequences f and g gives the values from 1 to $|T^iT'|$. So we merge f and g, during which we compute the Φ' values sequentially by Lemma 11. A pseudo-code of the algorithm is shown in Figure 4.

Thus, we have the following lemma.

j	$f(j)$	$\Phi[j]$	$f(\Phi[j])$	j	$g(j)$	$g(j+1)$
0		4				
1	1	3	$\underline{4}$			
				3	2	$(\underline{6})$
2	3	5	$\underline{7}$			
3	4	0	$\underline{0}$			
				2	5	$\underline{2}$
4	6	2	$\underline{3}$			
5	7	1	$\underline{1}$			
				1	8	$\underline{5}$

Fig. 3. Functions f and g. The $f(\Phi[j])$ and $g(j+1)$ values (underlined) form Φ' when we merge f and g. Note that $(\underline{6})$ denotes the special case of setting $\Phi'[g(j)]$ to $f(\Phi[0])$ when $j = \ell$.

$j_f \leftarrow 1, \quad j_g \leftarrow 1.$
for $t = 0, 1, .., |T^iT'|$
    if $t = g(\ell)$
        $\Phi'[t] \leftarrow f(\Phi[0])$;
    else if $t = f(j_f)$
        $\Phi'[t] \leftarrow f(\Phi[j_f])$, $\quad j_f$++;
    else $\Phi'[t] \leftarrow g(j_g + 1)$, $\quad j_g$++;

Fig. 4. Pseudo-code to construct Φ' sequentially.

Lemma 12. *Suppose that we have the lex-order of suf_j among the set $\{suf_k \mid 1 \leq k \leq \ell\}$ and the order(suf_j, T') for all $1 \leq j \leq \ell$, and the Ψ function of T'. Then the Ψ function for T^iT' can be constructed in $(H_0 + 2 + o(1))n$ bits in $O(n)$ time.*

In conclusion, we have:

Lemma 13. *Given T^i and the Ψ function of T'. Computing the Ψ function of T^iT' takes $O(\ell \log n + n)$ time.*

3.3 Overall Performance

Combining the results of Sections 3.1 and 3.2, we conclude the section with the following result:

Theorem 1. *Given a string T of length n, the Ψ function of T can be computed in $O(n \log n)$ time in $(H_0 + 2 + \epsilon)n$ bits space, for any $\epsilon > 0$.*

4 Space-efficient construction of FM-index

Apart from CSA, there is another compressed index for suffix array called FM-index [5], which has demonstrated its compactness in size while showing competitive performance in searching a pattern recently [4]. The index is particularly suited for text with small-sized alphabet. The core part of the construction algorithm involves the Burrows-Wheeler transformation [1], which is a common procedure used in various data compression algorithms, such as bzip2 [23].

Precisely, the Burrows-Wheeler transformation transforms a text T of length n into another text W, where W is shown to be compressible in terms of the empirical entropy of T [18]. The transformed text W is defined such that $W[i] = T[\mathrm{SA}[i]-1]$ if $\mathrm{SA}[i] > 0$, and $W[i] = \$$ if $\mathrm{SA}[i] = 0$.

Given the Ψ of T, we observe that for any p, $\mathrm{SA}[\Psi^k[p]] = \mathrm{SA}[p] + k$ [22]. Now, by setting $p = \Psi[0] = SA^{-1}[0]$, and computing $\Psi^k[p]$ iteratively for $k = 1, 2, \ldots, n$, we obtain the values of $SA[\Psi^k[p]] = k$. Immediately, we can set $W[\Psi^k[p]] = T[k-1]$. Since each computation of Ψ takes $O(1)$ time, W can be constructed in $O(n)$ time.

Thus, we have the following theorem.

Theorem 2. *Given the text T and the Ψ function of T, the Burrows-Wheeler transformation on T can be output directly in $O(n \log |\Sigma|)$ bits space and in $O(n)$ time.*

Once the Burrows-Wheeler transformation is completed, the remaining steps for the construction of FM-index can be done in $O(n)$ time using negligible space in addition to the output index. Thus, we have the following result:

Theorem 3. *Given the text T and the Ψ function of T, the FM-index of T can be constructed in $O(n)$ time, using $O(n \log |\Sigma|)$ bits in addition to the output index.*

5 Concluding Remarks

We have presented an algorithm that constructs CSA in $(H_0 + 2 + \epsilon)n$ bits space, for some fixed $\epsilon > 0$. The running time is $O(n \log n)$, which is independent of the alphabet size. In contrast, the fastest known algorithm using comparable space requires $O(|\Sigma| n \log n)$ time.

We have also given the first algorithm for constructing FM-index using limited working space. We show that once CSA is constructed, we can build FM-index from CSA in $O(n)$ time without extra working storage.

In the literature, there is an algorithm that constructs CSA in $O(n \log\log |\Sigma|)$ time, but the working space is considerably increased to $O(n \log |\Sigma|)$ bits [11]. An interesting open problem is: Can we construct CSA in $o(n \log n)$ time, while using only $O((H_0 + 1)n)$ bits of working 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 and J. I. Munro. Efficient Suffix Trees on Secondary Storage. In *Proc. ACM-SIAM SODA*, pages 383–391, 1996.
3. M. Farach. Optimal Suffix Tree Construction with Large Alphabets. In *Proc. IEEE FOCS*, pages 137–143, 1997.
4. P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proc. ACM-SIAM SODA*, pages 269–278, 2001.
5. P. Ferragine and G. Manzini. Opportunistic Data Structures with Applications. In *Proc. IEEE FOCS*, pages 390–398, 2000.
6. D. Graff and K. Chen. Chinese Gigaword, 2003. `http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC2003T09`.
7. R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. In *Proc. ACM STOC*, 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. Manuscript, 2001.
9. D. A. Grossman and O. Frieder. *Information Retrieval: Algorithms and Heuristics*. Kluwer Academic Publishers, Boston, 1998.
10. D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, New York, 1997.
11. W. K. Hon, K. Sadakane, and W. K. Sung. Breaking a Time-and-Sapce Barrier in Constructing Full-Text Indices. In *Proc. IEEE FOCS*, 2003. To appear.
12. E. Hunt, M. P. Atkinson, and R. W. Irving. A database index to large biological sequences. In *Proc. VLDB*, pages 410–421, 2000.
13. P. Ko and S. Aluru. Space Efficient Linear Time Construction of Suffix Arrays. In *Proc. CPM*, pages 200–210, 2003.
14. S. Kurtz. Reducing the Space Requirement of Suffix Trees. *Software Practice and Experiences*, 29:1149–1171, 1999.
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 *Proc. COCOON*, pages 401–410, 2002.
16. J. Larsson and K. Sadakane. Faster Suffix Sorting. Technical Report Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1-43/(1999), Lund University, 1999.
17. U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
18. G. Manzini. An Analysis of the Burrows-Wheeler Transform. *Journal of the ACM*, 48(3):407–430, 2001.
19. E. M. McCreight. A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
20. T. H. Ong and H. Chen. Updateable PAT-Tree Approach to Chinese Key Phrase Extraction using Mutual Information: A Linguistic Foundation for Knowledge Management. In *Proceedings of Asian Digital Library Conference*, 1999.
21. R. F. Rice. Some practical universal noiseless coding techniques. Technical Report JPL-79-22, Jet Propulsion Laboratory, Pasadena, California, 1979.
22. K. Sadakane. New Text Indexing Functionalities of the Compressed Suffix Arrays. *Journal of Algorithms*, in press.
23. J. Seward. The `bzip2` and `libbzip2` official home page, 1996. `http://sources.redhat.com/bzip2/`.
24. S. Shimozono, H. Arimura, and S. Arikawa. Efficient Discovery of Optimal Word Association Patterns in Large Text Databases. *New Generation Computing*, 18:49–60, 2000.