# A Space and Time Efficient Algorithm for Constructing Compressed Suffix Arrays *

Wing-Kai Hon[†] Tak-Wah Lam[†] Kunihiko Sadakane[‡]
Wing-Kin Sung[§] Siu-Ming Yiu[†]

**Abstract**

With the first human DNA being decoded into a sequence of about 2.8 billion characters, many biological research has been centered on analyzing this sequence. Theoretically speaking, it is now feasible to accommodate an index for human DNA in the main memory so that any pattern can be located efficiently. This is due to the recent breakthrough on compressed suffix arrays, which reduces the space requirement from $O(n \log n)$ bits to $O(n)$ bits for indexing a text of $n$ characters. However, constructing compressed suffix arrays is still not an easy task because we still have to compute suffix arrays first and need a working memory of $O(n \log n)$ bits (i.e., more than 13 Gigabytes for human DNA). This paper initiates the study of constructing compressed suffix arrays directly from the text. The main contribution is a construction algorithm that uses only $O(n)$ bits of working memory, and the time complexity is $O(n \log n)$. Our construction algorithm is also time and space efficient for texts with large alphabets such as Chinese or Japanese. Precisely, when the alphabet size is $|\Sigma|$, the working space becomes $O(n(H_0 + 1))$ bits, where $H_0$ denotes the order-0 entropy of the text and it is at most $\log |\Sigma|$; for the time complexity, it remains $O(n \log n)$ which is independent of $|\Sigma|$.

## 1 Introduction

DNA sequences, which hold the code of life for living organisms, can be represented by strings over four characters A, C, G, and T. With the advance in bio-technology, the complete DNA sequences for a number of living organisms have been known. Even for human DNA, a draft which comprises about 2.8 billion characters, has been finished recently. This paper is concerned with data structures for indexing a DNA sequence so that searching for an arbitrary pattern

*Results in this paper have appeared in a preliminary form in the Proceedings of the 8th Annual International Computing and Combinatorics Conference, 2002 and the Proceedings of the 14th International Conference on Algorithms and Computation, 2003.

[†]Department of Computer Science, The University of Hong Kong, Hong Kong, {wkhon,twlam,smyiu}@csis.hku.hk. Research was supported in part by the Hong Kong RGC Grant HKU-7042/02E.

[‡]Department of Computer Science and Communication Engineering, Kyushu University, Japan, sada@csce.kyushu-u.ac.jp. Research was supported in part by the Grant-in-Aid of the Ministry of Education, Science, Sports and Culture of Japan.

[§]School of Computing, National University of Singapore, Singapore, ksung@comp.nus.edu.sg. Research was supported in part by the NUS Academic Research Grant R-252-000-119-112.

can be performed efficiently. Such tools find applications to many biological research activities on DNA, such as gene hunting, promoter consensus identification, and motif finding. Unlike English text, DNA sequences do not have word boundaries; suffix trees [18] and suffix arrays [16] are the most appropriate solutions in the literature for indexing DNA. For a DNA sequence with $n$ characters, building a suffix tree takes $O(n)$ time, then a pattern $P$ can be located in $O(|P| + occ)$ time, where $occ$ is the number of occurrences. For suffix arrays, construction and searching takes $O(n)$ time and $O(|P| \log n + occ)$ time, respectively. Both data structures require $O(n \log n)$ bits; suffix array is associated with a smaller constant, though. For human DNA, the best known implementation of suffix tree and suffix array require 40 Gigabytes and 13 Gigabytes, respectively [13]. Such memory requirement far exceeds the capacity of ordinary computers. Existing approaches for indexing human DNA include (1) using supercomputers with large main memory [22]; and (2) storing the indexing data structure in the secondary storage [2, 11]. The first approach is expensive and inflexible, while the second one is slow. As more and more DNA are decoded, it is vital that individual biologists can eventually analyze different DNA sequences efficiently with their PCs.

Recent breakthrough results in compressed suffix arrays, namely, the Compressed Suffix Arrays (CSA) proposed by Grossi and Vitter [7], and the FM-index proposed by Ferragina and Manzini [3], shed light on this direction. It is now feasible to store a compressed suffix array of human DNA in the main memory, which occupies only $O(n)$ bits.[1] Pattern search can still be performed efficiently, the time complexity increases only by a factor of $\log n$. For human DNA, a compressed suffix array occupies about 2 Gigabytes. Nowadays a PC can have up to 4 Gigabytes of main memory and can easily accommodate such a data structure. For the performance of CSA and FM-index in practice, one can refer to [4, 6, 9].

Theoretically speaking, a compressed suffix array can be constructed using $O(n)$ time; however, the construction process requires much more than $O(n)$ bits of working memory. Among others, the original suffix array has to be built first, taking up at least $n \log n$ bits. In the context of human DNA, the working memory for constructing a compressed suffix array is at least 40 Gigabytes [22], far exceeding the capacity of ordinary PCs. This motivates us to investigate whether we can construct a compressed suffix array using $O(n)$ bits of memory, perhaps with a slight increase in construction time. The space requirement means construction directly from DNA sequences. This paper provides the first algorithm of such a kind, showing that the basic form of the CSA—the $\Psi$ array—can be built in a space and time efficient manner, which can then be easily converted to the FM-index. In addition, our construction algorithm can be used to construct the hierarchical CSA [7].

Our construction algorithm for the $\Psi$ array also works well for texts without word boundary, such as Chinese or Japanese, whose alphabet consists of at least a few thousand characters. Precisely, for a text with an alphabet $\Sigma$, our algorithm requires $O(n(H_0 + 1))$ bits of working memory, where $H_0$ denotes the order-0 entropy of the text and it is at most $\log |\Sigma|$. The time complexity is $O(n \log n)$, which is independent of $|\Sigma|$.

Experiments show that for human DNA, our space-efficient algorithm for the $\Psi$ array can run on a PC with 3 Gigabytes of memory and takes about 21 hours [9], which is only about three times slower than the original algorithm implemented on a supercomputer with 64 Gigabytes of main memory to accommodate the suffix array [22].

[1] In general, for a text over an alphabet $\Sigma$, CSA occupies $nH_k + o(n)$ bits [7, 5] and FM-index requires $O(nH_k) + o(n)$ bits [3], where $H_k$ denotes the $k$-th entropy of the text and $H_k$ is upper bounded by $\log |\Sigma|$.

Remark: More recently, Hon et al. [10] have derived an alternative algorithm for constructing the $\Psi$ array, which runs in $O(n \log\log|\Sigma|)$ time; however, the space requirement is $O(n \log|\Sigma|)$, which is not preferred for texts with a large alphabet but with small entropy such as XML documents.

Technically speaking, our algorithm does not require much space other than that for storing the $\Psi$ array. This is based on an observation that the $\Psi$ arrays of two consecutive suffixes are very similar. Thus, we can build the entire $\Psi$ array directly from the text in an incremental 'character by character' manner. Exploiting this observation further, we can speed up the construction by processing more characters each time, yielding a 'segment by segment' algorithm.

The rest of this paper is organized as follows. Section 2 reviews the suffix arrays and the $\Psi$ array. Section 3 relates the $\Psi$ arrays between two consecutive suffixes, thereby giving a taste of constructing the $\Psi$ array in a 'character by character' manner. Section 4 details the 'segment by segment' construction algorithm for the $\Psi$ array, while Section 5 discusses the construction of the hierarchical CSA and the conversion of $\Psi$ into the FM-index in a space-efficient manner.

# 2 Preliminaries

In this section, we review the definitions of suffix arrays and the basic form of the Compressed Suffix Arrays (CSA), which is called the $\Psi$ array. Also, we introduce some notations to be used throughout the paper. In addition, some simple observations on the $\Psi$ array are presented.

Let $T$ be a text over an alphabet $\Sigma$. Throughout this paper, we assume that $T$ is given a special character \$ at the end, where \$ is not in $\Sigma$ and is lexicographically smaller than all characters in $\Sigma$. Let $n$ be the number of characters (including \$) in $T$. $T$ is assumed to be stored in an array $T[0..n-1]$. For any integer $i \in [0, n-1]$, we denote

- $T[i]$ as the $(i+1)$-th character of $T$ from the left (thus, $T[n-1] = \$$); and
- $T_i$ as the suffix of $T$ starting from the position $i$; that is, $T_i = T[i..n-1] = T[i]T[i+1]\dots T[n-1]$.

Furthermore, let $\mathcal{S}(T)$ denote the set of all suffixes of $T$, $\{T_0, T_1, \cdots, T_{n-1}\}$.

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

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

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

Figure 1: The suffix array and the $\Psi$ array of $acaaccg\$$

**Suffix Arrays:** A suffix array [16] of $T$, denoted SA$[0..n-1]$, is a sorted sequence of the suffixes of $T$. Formally, SA$[i]$ denotes the starting position of the $(i+1)$-th smallest suffix of

$T$. In other words, according to the lexicographical order, $T_{\text{SA}[0]} < T_{\text{SA}[1]} < \ldots < T_{\text{SA}[n-1]}$. See Figure 1 for an example. Note that $\text{SA}[0] = n - 1$. Each $\text{SA}[i]$ can be represented in $\lceil \log n \rceil$ bits, and the suffix array can be stored using $n\lceil \log n \rceil$ bits.[2] Given a text $T$ together with the suffix array $\text{SA}[0..n-1]$, 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 [16].

For every integer $i \in [0, n-1]$, define $\text{SA}^{-1}[i]$ to be the integer $j$ such that $\text{SA}[j] = i$. Intuitively, $\text{SA}^{-1}[i]$ denotes the rank of $T_i$ among the suffixes of $T$, which is the number of suffixes of $T$ lexicographically smaller than $T_i$. We use the notation $\text{Rank}(X, \mathcal{S})$ to denote the rank of $X$ among a set of strings $\mathcal{S}$. Thus, $\text{SA}^{-1}[i] = \text{Rank}(T_i, \mathcal{S}(T))$.

**The Basic Form of the CSA:** Based on SA and $\text{SA}^{-1}$, the basic form of the CSA of a text $T$ is an array $\Psi[0..n-1]$ defined as follows [7]: $\Psi[i] = \text{SA}^{-1}[\text{SA}[i] + 1]$ for $i = 1, 2, \ldots, n-1$, whereas $\Psi[0]$ is defined as $\text{SA}^{-1}[0]$. In other words, if $T_k$ is the suffix with rank $i$, $\Psi[i]$ is the rank of the suffix $T_{k+1}$. See Figure 1 for an example. It is worth-mentioning that $\Psi$ can be used to recover $\text{SA}^{-1}$ iteratively: $\text{SA}^{-1}[1] = \Psi[\Psi[0]]$, $\text{SA}^{-1}[2] = \Psi[\Psi[\Psi[0]]]$, ..., etc.

Note that $\Psi[0..n-1]$ contains $n$ integers. A trivial way to store the array requires $n\lceil \log n \rceil$ bits, using the same space as SA. Nevertheless, $\Psi[1..n-1]$ can be decomposed into $|\Sigma|$ strictly increasing sequences, which allows it to be stored succinctly. See Figure 1 for an illustration. This increasing property is based on the following lemmas.

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

**Proof:** Note that $i < j$ if and only if $T_{\text{SA}[i]} < T_{\text{SA}[j]}$. This implies that if $i < j$ and $T[\text{SA}[i]] = T[\text{SA}[j]]$, $T_{\text{SA}[i]+1} < T_{\text{SA}[j]+1}$. Equivalently, we have $T_{\text{SA}[\Psi[i]]} < T_{\text{SA}[\Psi[j]]}$. Thus, $\Psi[i] < \Psi[j]$ and the lemma follows. □

For each character $c$, let $\alpha(c)$ be the number of suffixes starting with a character lexicographically smaller than $c$, and let $\#(c)$ be the number of suffixes starting with $c$.

**Corollary 1** *For each character $c$, $\Psi[\alpha(c)..\alpha(c)+\#(c)-1]$ gives a strictly increasing sequence.*

**Proof:** For any character $c$, $T[\text{SA}[\alpha(c)]] = T[\text{SA}[\alpha(c)+1]] = \cdots = T[\text{SA}[\alpha(c)+\#(c)-1]] = c$. By Lemma 1, $\Psi$ is strictly increasing in $\Psi[\alpha(c)..\alpha(c)+\#(c)-1]$. □

Based on the above increasing property, Grossi and Vitter [8] devised a scheme to store $\Psi$ of a binary text in $O(n)$ bits. In fact, this scheme can be easily extended for storing $\Psi$ of a general text, taking $O(n(H_0 + 1))$ bits, where $H_0 \leq \log|\Sigma|$ is the order-0 entropy of the text $T$. Details are as follows. For each character $c$, the sequence $\Psi[\alpha(c)..\alpha(c) + \#(c) - 1]$ is represented using Rice code [20]. That is, each $\Psi[i]$ in the sequence is divided into two parts $q_i$ and $r_i$, where $q_i$ is the first (or most significant) $\lfloor \log \#(c) \rfloor$ bits, and $r_i$ is the remaining $\lceil \log n \rceil - \lfloor \log \#(c) \rfloor$ bits, which is at most $\lceil \log(n/\#(c)) \rceil + 1$ bits. The $r_i$'s are stored explicitly in an array of size $\#(c)(\lceil \log(n/\#(c)) \rceil + 1)$ bits. For the $q_i$'s, since they form a monotonic increasing sequence bounded by 0 and $\#(c) - 1$, we store $q_{\alpha(c)}$, and the difference values $q_{i+1} - q_i$ for $i \in [\alpha(c), \alpha(c)+\#(c)-2]$ using unary codes,[3] which requires $2\#(c)$ bits. In total, the space required

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

[3] The unary code for an integer $x \geq 0$ is encoded as $x$ `0`'s followed by a `1`.

is at most $\sum_{c\in\Sigma} \#(c)(\lceil\log(n/\#(c))\rceil + 3)$. By definition, $nH_0$ is equal to $\sum_{c\in\Sigma} \#(c)\log(n/\#(c))$, the total space is thus at most $(H_0+4)n$ bits.

Based on the above discussion, we have the following lemma.

**Lemma 2** *The $\Psi$ array can be represented using $O(n(H_0+1))$ bits. If we can enumerate the values of $\Psi[i]$ sequentially, this representation can be constructed directly using $O(n)$ time without extra working space.*

With the above representation scheme, each $\Psi$ value can be retrieved in $O(1)$ time by using the following auxiliary data structures. They include: (1) Raman et al.'s dictionary (Lemma 2.3 in [19]) on the values of $\alpha(c)$ for all $c$ in $\Sigma$, which supports for each $c$ finding $\alpha(c)$ in $O(1)$ time, and supports for each $i$ finding the largest $c$ with $\alpha(c) \le i$ in $O(1)$ time; (2) the unary encoded $q_i$'s for $c = 1, 2, \ldots, |\Sigma|$ are stored consecutively as a bit-vector $B$ of at most $2n$ bits, and we create Jacobson's data structure [12] on $B$ to support $O(1)$-time `rank` and `select` queries; (3) Raman et al.'s dictionary on the pointers to the arrays of $r_i$'s, which supports for each $c$ an $O(1)$-time retrieval of the corresponding pointer.

To find $\Psi[i]$, we compute the largest $c$ such that $\alpha(c) \le i$. Then, we know that $\Psi[i]$ is within the strictly increasing sequence of $\Psi[\alpha(c)..\alpha(c)+\#(c)-1]$. Next, $q_i$ can be obtained by counting the number of 0's between the $\alpha(c)$-th 1 and the $(i+1)$-th 1 in $B$. To obtain $r_i$, we compute $\#(c) = \alpha(c+1) - \alpha(c)$, following the pointers for the array of $r_i$'s for $c$, and retrieve the $(i - \alpha(c) + 1)$-th entry in the array (knowing that each entry occupies $\lceil\log(n/\#(c))\rceil + 1$ bits). Each of the above step can be computed in $O(1)$ time, so that the time follows.

For the space complexity, the Raman et al.'s dictionaries for $\alpha(c)$ values and the pointers take $\log\binom{n+|\Sigma|}{|\Sigma|} + o(n)$ bits and $\log\binom{n(H_0+4)+|\Sigma|}{|\Sigma|} + o(n(H_0+1))$ bits, respectively, while the Jacobson's data structure has size $o(n)$ bits. Thus, the auxiliary structures have a total size of $O(n(H_0+1))$ bits. This gives the following lemma.

**Lemma 3** *The representation of $\Psi$ in Lemma 1 can be augmented with auxiliary data structures of total size $O(n(H_0+1))$ bits, so that any $\Psi$ value can be retrieved in $O(1)$ time.*

In the literature, there is another representation of the $\Psi$ array which, instead of viewing $\Psi$ as a set of $|\Sigma|$ increasing sequences, considers the $\Psi$ array as $|\Sigma|^k$ sets of $|\Sigma|$ increasing sequences and encode each set of increasing sequence independently using Rice code. The resulting data structure requires only $O(n(H_k+1))$ bits for storage when $k+1 \le \log_{|\Sigma|} n$, while supporting $O(1)$-time retrieval of any $\Psi$ value [5]. Nevertheless in the remaining paper, we shall assume the above $O(n(H_0+1))$-bit scheme for storing $\Psi$; that is, using the scheme of Lemma 2 for representing the $\Psi$ array, and augmenting it with the auxiliary data structures of Lemma 3.

# 3 The $\Psi$ Arrays of Two Consecutive Suffixes

This section serves as a warm up to the main algorithm presented in the next section. In particular, we investigate the relationship between the $\Psi$ arrays of two consecutive suffixes. Then, based on this relationship, we demonstrate an algorithm that constructs the $\Psi$ array for a text $T$, in an incremental manner. Since this algorithm is not the main result of this paper, we only give the high-level description. One can refer to [14] for the implementation details.

Let $T$ be a string with $n$ characters. We assume that $T$ is represented by an array $T[0..n-1]$ and $T[n-1] = \$$. Let $\mathrm{SA}_T$ and $\Psi_T$ be the suffix array and $\Psi$ array of $T$, respectively.

Suppose that we are given the $\Psi$ array of $T$, and we want to construct the $\Psi$ array for a longer text $T' = cT$, where $c$ is a character. Let $\mathrm{SA}_{T'}$ and $\Psi_{T'}[0..n]$ denote the suffix array and the $\Psi$ array of $T'$, respectively. To see the relationship between the $\Psi$ arrays of $T$ and $T'$, we first show that the suffix array of $T'$ can be easily obtained from that of $T$.

Recall that $\mathrm{SA}_T$ is a sequence of the starting positions of the suffixes of $T$, sorted according to their ranks. Except $T'$ itself, $T'$ shares all its suffixes with $T$; thus, $\mathrm{SA}_{T'}$ has exactly one more entry than $\mathrm{SA}_T$, which is due to the suffix $T'$. Intuitively, to obtain $\mathrm{SA}_{T'}$, we can insert the suffix $T'$ (which is represented by the starting position 0) into $\mathrm{SA}_T$. of $T$. Let $x = \mathrm{Rank}(T', \mathcal{S}(T))$. $T'$ should be inserted between $\mathrm{SA}_T[x-1]$ and $\mathrm{SA}_T[x]$. Also, since a character is added to the beginning of $T$, we increment every entry of $\mathrm{SA}_T$ by 1 to reflect the change in their starting position. Thus, we have the following lemma.

**Lemma 4** *Let $x = \mathrm{Rank}(T', \mathcal{S}(T))$. Then,*

$$\mathrm{SA}_{T'}[i] = \begin{cases} \mathrm{SA}_T[i] + 1 & \text{if } 0 \le i \le x-1 \\ 0 & \text{if } i = x \\ \mathrm{SA}_T[i-1] + 1 & \text{if } i \ge x+1 \end{cases}$$

Based on Lemma 4, we observe the relationship between the $\Psi$ arrays of $T$ and $T'$ as follows.

**Lemma 5** *Let $x = \mathrm{Rank}(T', \mathcal{S}(T))$. Then,*

- $\Psi_{T'}[0] = x$;
- *for* $1 \le i < x$, $\Psi_{T'}[i] = \begin{cases} \Psi_T[i] & \text{if } \Psi_T[i] < x \\ \Psi_T[i] + 1 & \text{if } \Psi_T[i] \ge x \end{cases}$;
- *for* $i = x$, $\Psi_{T'}[i] = \begin{cases} \Psi_T[0] & \text{if } \Psi_T[0] < x \\ \Psi_T[0] + 1 & \text{if } \Psi_T[0] \ge x \end{cases}$;
- *for* $x < i \le n$, $\Psi_{T'}[i] = \begin{cases} \Psi_T[i-1] & \text{if } \Psi_T[i-1] < x \\ \Psi_T[i-1] + 1 & \text{if } \Psi_T[i-1] \ge x \end{cases}$.

The above lemma suggests that we can compute $\Psi_{T'}$ from $\Psi_T$ as follows.

1. Compute $x =$ the rank of $T'$ among all suffixes of $T$.
2. Set $\Psi_{T'}[0] = x$.
3. For $1 \le i \le n$, set $\Psi_{T'}[i] = \begin{cases} \Psi_T[i] & \text{if } i < x \\ \Psi_T[0] & \text{if } i = x \\ \Psi_T[i-1] & \text{if } i > x \end{cases}$
4. For each $1 \le i \le n$, if $\Psi_{T'}[i] \ge x$, increment $\Psi_{T'}[i]$ by 1.

To build the $\Psi$ array for a text $T$ of length $n$ starting from scratch, we can execute the above algorithm repeatedly, constructing the $\Psi$ arrays for the suffixes $T_{n-1}, T_{n-2}, \cdots, T_0$ incrementally. Each such execution can be implemented in $O(n)$ time. Thus, we can construct $\Psi_T$ for $T[0..n-1]$ using $O(n^2)$ time. In the next section, we will present how to improve the construction time to $O(n \log n)$. The idea is that, instead of updating the $\Psi$ array every time a character is added, we collectively perform the update for every 'segment'. This gives an incremental algorithm which processes the text in a 'segment by segment' manner.

# 4 Incremental Algorithm for Constructing the $\Psi$ Array

In this section, we show how to compute $\Psi[0..n-1]$ for the text $T$ incrementally, in a 'segment by segment' manner. To do so, we first partition the text into $\lceil n/\ell \rceil$ consecutive segments $T^1, T^2, \ldots, T^{\lceil n/\ell \rceil}$, where $\ell = \Theta(n/\log n)$; each segment, except the last one, contains $\ell$ characters, i.e., $T^i$ refers to the string represented by $T[(i-1)\ell..i\ell-1]$. The algorithm builds the $\Psi$ array of $T$ incrementally, starting with that of $T^{\lceil n/\ell \rceil}$, and then constructs the $\Psi$ array of $T^{\lceil n/\ell \rceil - 1}T^{\lceil n/\ell \rceil}$ and so on. Eventually the $\Psi$ array of $T^1T^2 \ldots T^{\lceil n/\ell \rceil} = T$ is constructed. Below, we show that the construction time required for each segment is $O(\ell \log n + n) = O(n)$ time, and the overall time is $O(n \log n)$, which is independent of $|\Sigma|$. For the space requirement, it is $O(n(H_0 + 1))$ bits.

Recall from the last section that, when we construct the $\Psi$ array character by character, the key point is to compute the rank of the newly added suffix among the existing ones, and alter the existing $\Psi$ array accordingly. Indeed, when we construct the $\Psi$ array segment by segment, the idea is similar. To cater for a new segment, we again compute the ranks of all newly added suffixes among the existing ones. It is obvious that these ranks represent the positions in the suffix array where the new suffixes are to be inserted. Accordingly the existing $\Psi$ array needs to be expanded in order to insert the new suffixes. However, knowing such rank is not sufficient. We also need the rank of the new suffixes among themselves. Details are as follows.

Consider any $i \in [1, \lceil n/\ell \rceil - 1]$. Let $B$ denote the string $T^{i+1}T^{i+2} \cdots T^{\lceil n/\ell \rceil}$. Suppose that we have built $\Psi_B$, the $\Psi$ array of $B$. Let $A = T^iB$. Adding $T^i$ to $B$ introduces $\ell$ new suffixes; we call them the *long suffixes* of $A$. The set of the long suffixes are referred to as $\mathcal{LS}(A)$. Other suffixes of $A$ are also suffixes of $B$, we call them the *short suffixes*. Note that $\mathcal{S}(A) = \mathcal{S}(B) \cup \mathcal{LS}(A)$. To determine the rank of a long suffix $x$ among $\mathcal{S}(A)$, we can compute the rank of $x$ among $\mathcal{S}(B)$ and the rank of $x$ among $\mathcal{LS}(A)$, and then sum them up.

**Fact 1** *Let $x$ be a a long suffix of $A$ (i.e., $x = A_k$ for some $k \in [0, \ell-1]$). Then* $\text{Rank}(x, \mathcal{S}(A)) = \text{Rank}(x, \mathcal{LS}(A)) + \text{Rank}(x, \mathcal{S}(B))$.

Once the rank of the long suffixes among $\mathcal{S}(A)$ is known, we can also compute the rank of each short suffix among $\mathcal{S}(A)$ by simply adjusting the rank of a short suffix among $\mathcal{S}(B)$ according to distribution of the long suffixes. To speed up the computation, we exploits a data structure that supports in $O(1)$ time the rank and select operations.

In Sections 4.1 and 4.2, we show how to compute $\text{Rank}(x, \mathcal{LS}(A))$ and $\text{Rank}(x, \mathcal{S}(B))$ for every long suffix $x$, respectively. In addition, we describe how to store them in a space efficient way while allowing fast retrieval. In Section 4.3, we give the details of constructing $\Psi_A$ from

$\Psi_B$, and show that the $\Psi$ array of $T$ can be constructed in $O(n \log n)$ time using $O(n(H_0+1))$ bits.

Before moving to the details of the incremental construction, we give the details for building the first $\Psi$ array (i.e., the $\Psi$ for $T^{\lceil n/\ell \rceil}$). Note that $T^{\lceil n/\ell \rceil}$ contains at most $\ell$ characters and a brute force approach for constructing $\Psi$ does not use too much space. Precisely, this $\Psi$ can be obtained easily in $O(\ell \log \ell)$ time using $3\ell \lceil \log n \rceil$ bits of space as follows. We use three arrays of $\ell \lceil \log n \rceil$ bits for storing the SA, $\mathrm{SA}^{-1}$ and $\Psi$ of $T^{\lceil n/\ell \rceil}$ explicitly. First, we compute the SA for $T^{\lceil n/\ell \rceil}$ by suffix sorting, which takes $O(\ell \log \ell)$ time using $\ell \lceil \log n \rceil$ bits in addition to that for storing SA [15]. Afterwards, the $\mathrm{SA}^{-1}$ can be computed in $O(\ell)$ time. When both SA and $\mathrm{SA}^{-1}$ are available, we can construct the representation of $\Psi$ (under the scheme of Lemma 1) in $O(\ell)$ time. For the auxiliary data structures (under the scheme of Lemma 2), they are computed in $O(\ell + |\Sigma|)$ time: (1) The Raman et al.'s dictionary for the $\alpha(c)$ values are constructed by examining the SA array sequentially, using $O(\ell + |\Sigma|)$ time; (2) the two remaining data structures are computed along with the representation of $\Psi$, taking an extra $O(\ell)$ time.

## 4.1 Rank of long suffixes among themselves

This section describes how to compute the rank of the $\ell$ long suffixes of $A$ among themselves (i.e., suffixes in $\mathcal{LS}(A)$). A straightforward method is to sort the suffixes of $A$ and then determine the rank of every suffix of $A$ among themselves. However, this requires $O(n \log n)$ time when $|A| = O(n)$ [15]. In fact, when given $\Psi_B$, a simple observation shows that it suffices to perform suffix sorting on the prefix $A[0..2\ell-1]$ only, and the time is reduced to $O(\ell \log \ell)$. The idea is as follows: If the first $\ell$ characters of two suffixes (say, $A_i$ and $A_j$) in $\mathcal{LS}(A)$ are different, their relative order can be decided immediately; otherwise we resolve their relative order by comparing their suffixes starting at the $(\ell+1)$-th character, which are exactly the suffixes of $B$ starting at position $i$ and $j$ (i.e., $B_i$ and $B_j$). Note that the relative order of $B_i$ and $B_j$ can be deduced from $\Psi_B$. More precisely, define $P$ and $Q$ to be two arrays of $\ell$ integers such that for all $k \in [0, \ell-1]$,

- $P[k]$ is the rank of $A_k$ among $\mathcal{LS}(A)$ when only the first $\ell$ characters are considered;
- $Q[k]$ is the rank of $B_k$ among $\mathcal{S}(B)$.

Let $(p_1, q_1)$ and $(p_2, q_2)$ be two tuples. We say $(p_1, q_1)$ is smaller than $(p_2, q_2)$ if (i) $p_1 < p_2$ or (ii) $p_1 = p_2$ and $q_1 < q_2$. For any tuple $(p, q)$ among a set of tuples $S$, the rank of $(p, q)$ is the number of tuples in $S$ that is smaller than $(p, q)$. Then, we have the following fact.

**Fact 2** *Consider the $\ell$ tuples $(P[k], Q[k])$ for all $k \in [0, \ell-1]$. For any long suffix $A_h$,* $\mathrm{Rank}(A_h, \mathcal{LS}(A))$ *is equal to the rank of $(P[h], Q[h])$ among these $\ell$ tuples.*

Suppose that $\Psi_B$ is given. Below we give the details of computing the arrays $P$ and $Q$. Then, we make use of the above fact to compute the rank of the long suffixes of $A$ among themselves. The results are stored in an array called $M$. Details are as follows:

**Step 1: Computing $P$.** To sort the $\ell$ long suffixes of $A$ according to their first $\ell$ characters, we focus on the substring $A[0..2\ell-1]$ and apply the suffix sorting algorithm of Larsson and Sadakane [15] for $\lceil \log \ell \rceil$ rounds, which can figure out the order of the suffixes according to the first $\ell$ characters. Then, for each $k \in [0..\ell-1]$, we extract the rank of $A_k$ and store it into $P[k]$. The time required is $O(\ell \log \ell)$.

**Step 2: Computing $Q$.** For any $k \in [0..\ell-1]$, $Q[k] = \text{Rank}(B_k, \mathcal{S}(B))$, which is equal to $\text{SA}_B^{-1}[k]$. By definition, $\text{SA}_B^{-1}[0] = \Psi_B[0]$, $\text{SA}_B^{-1}[1] = \Psi_B[\Psi_B[0]]$, and in general, $\text{SA}_B^{-1}[k] = \Psi_B^{(k+1)}[0]$. Thus, we can compute $Q$ by evaluating $\Psi^{(k)}[0]$ iteratively for $k = 1, \ldots, \ell$. The time required is $O(\ell)$.

**Step 3: Sorting.** Consider the tuples $(P[k], Q[k])$ for all $k \in [0, \ell-1]$. Perform the sorting on these tuples in $O(\ell \log \ell)$ time, Then, for each $k \in [0, \ell-1]$, $M[k]$ is the order of $\text{Rank}(A_k, \mathcal{LS}(A))$.

*Time and space requirement:* Steps 1-3 altogether require $O(\ell \log \ell)$ time. As to be shown later, we will also need the inverse of $M$, denoted $M^{-1}$, which can be computed from $M$ in $O(\ell)$ time. Note that $M$ and $M^{-1}$ each require $\ell \lceil \log n \rceil$ bits, and the above steps require an additional working space of $2\ell \lceil \log n \rceil$ bits (for storing $P$ and $Q$). The total space requirement is $4\ell \lceil \log n \rceil$ bits.

## 4.2 Rank of long suffixes among $\mathcal{S}(B)$

This section shows that if $\Psi_B$ is given, then the rank of the $\ell$ long suffixes of $A$ among all suffixes of $B$ can be computed in $O(\ell \log n + n)$ time. Apart from $\Psi_B$, the space required is $\ell \log n$ bits, which is essentially needed for storing the output.

For any character $c$, let $\#_B(c)$ denote the number of suffixes of $B$ starting with $c$, and let $\alpha_B(c)$ denote the number of suffixes of $B$ whose starting character is lexicographically smaller than $c$. Note that these numbers are stored in the auxiliary data structure of $\Psi_B$, and each of them can be retrieved in $O(1)$ time. All suffixes of $B$ starting with $c$ have a rank in the range $[\alpha_B(c), \alpha_B(c) + \#_B(c) - 1]$, which is denoted $R_B(c)$ below. The following lemma shows how to determine rank incrementally, i.e., how to derive $\text{Rank}(cX, \mathcal{S}(B))$ from $\text{Rank}(X, \mathcal{S}(B))$ for any string $X$ and character $c$.

**Lemma 6** *Consider any string $X$ and any character $c$. Let $\mathcal{H}$ denote the set $\{r \in R_B(c) \mid \Psi_B[r] < \text{Rank}(X, \mathcal{S}(B))\}$. Then,*

$$\text{Rank}(cX, \mathcal{S}(B)) = \begin{cases} \alpha_B(c) & \text{if } \mathcal{H} \text{ is empty} \\ 1 + \max\{r \mid r \in \mathcal{H}\} & \text{otherwise} \end{cases}$$

**Proof:** First, we claim that $\mathcal{H}$ stores the rank of all those suffixes of $B$ which have $c$ as the first character, and which are lexicographically smaller than $cX$. The reason is as follows: Consider any suffix of $B_i$ whose first character is $c$. Let $r$ be its rank among $S(B)$. Note that $r$ is within $R_B(c)$. If $B_i < cX$, then $B_{i+1} < X$. Denote the rank of $B_{i+1}$ as $r'$. Then $r' < \text{Rank}(X, \mathcal{S}(B))$. On the other hand, by definition, $\Psi_B[r] = \text{SA}_B^{-1}[\text{SA}_B[r]+1] = r'$ (where $\text{SA}_B$ and $\text{SA}_B^{-1}$ denotes the suffix array of $B$ and its inverse). Therefore, $\Psi_B[r] < \text{Rank}(X, \mathcal{S}(B))$. Reversing the argument, we can show that for every $r \in R_B(c)$ with $\Psi_B[r] < \text{Rank}(X, \mathcal{S}(B))$, the suffix of $B$ with rank $r$ (i.e., $B_{\text{SA}_B[r]}$) is lexicographically smaller than $cX$. Thus, the claim follows.

We are now ready to prove the lemma. If $\mathcal{H}$ is empty, any suffix of $B$ starting with character $c$ is lexicographically larger than or equal to $cX$. Then, $\text{Rank}(cX, \mathcal{S}(B))$ is equal to the rank of the single character $c$ among $\mathcal{S}(B)$, which is $\alpha_B(c)$. If $\mathcal{H}$ is not empty, $\text{Rank}(cX, \mathcal{S}(B)) = \alpha_B(c) + |\mathcal{H}|$. By Corollary 1, $\Psi_B[r]$ is strictly increasing for $r \in R_B(c)$, and $\mathcal{H}$ is equal to

$\{\alpha_B(c), \alpha_B(c)+1, \ldots, \alpha_B(c)+|\mathcal{H}|-1\}$. Thus, $\max\{r \mid r \in \mathcal{H}\} = \alpha_B(c)+|\mathcal{H}|-1$, and the lemma follows. □

Based on the above lemma, we can compute the required rank in a backward manner as follows. The result is stored in an array $L[0..\ell-1]$ such that $L[k] = \mathrm{Rank}(A_k, \mathcal{S}(B))$ for all $k \in [0, \ell-1]$.

> For $k = \ell - 1$ down to 0, compute $L[k]$ as follows: Let $c = A[k]$. The suffix $A_k$ can be expressed as $cA_{k+1}$. Note that $\mathrm{Rank}(A_{k+1}, \mathcal{S}(B))$ has been computed and stored in $L[k+1]$.[4] To compute $L[k]$, we find the maximum $r \in R_B(c)$ satisfying $\Psi_B[r] < L[k+1]$. Since $\Psi_\mathrm{B}$ is strictly increasing in the range $R_B(c)$, we can use a binary search to find the maximum $r$; this requires $O(\log n)$ time. If $r$ exists, we set $L[k]$ to be $r+1$; otherwise, we set $L[k]$ to be $\alpha_B(c)$.

*Time and space requirement:* The time required for computing $L$ is $O(\ell \log n)$, and $L$ occupies $\ell\lceil\log n\rceil$ bits. Thus, the total time and total space required are both $O(n)$.

## 4.3 Computing $\Psi_A$

This section shows how to make use of the results of Sections 4.1 and 4.2 to compute $\Psi_A$ in $O(\ell \log n + n)$ time. For the space requirement, it takes $4\ell\lceil\log n\rceil + o(n)$ bits in addition to that for maintaining $\Psi_A$ and $\Psi_B$. Recall that the following three arrays are available.

1. An array $M$ such that $M[i]$ stores $\mathrm{Rank}(A_i, \mathcal{LS}(A))$.
2. An array $M^{-1}$, which is the inverse of $M$, such that $M^{-1}[i]$ stores the position of the suffix among $\mathcal{LS}(A)$ whose rank is $i$.
3. An array $L$ such that $L[i]$ stores $\mathrm{Rank}(A_i, \mathcal{S}(B))$.

By Fact 1, we can compute the rank of each long suffix $A_k$ (where $k \in [0, \ell-1]$) among $\mathcal{S}(A)$ by summing $M[k]$ and $L[k]$. For the short suffixes of $A$, their rank among $\mathcal{S}(A)$ can be figured out by adjusting their rank among $\mathcal{S}(B)$ according to distribution of the long suffixes. Precisely, let $m = |A|$, and define $V[0..m-1]$ to be a bit vector such that $V[i] = 1$ if the suffix of $A$ with rank $i$ is a long suffix, and $V[i] = 0$ otherwise. We need $V$ to support two types of efficient queries:

- $\mathtt{rank}_0(V, i)$ and $\mathtt{rank}_1(V, i)$ returns the number of 0's and 1's preceding $V[i]$, respectively.
- $\mathtt{select}_0(V, j)$ returns the position of the $j$-th 0 in $V$.

Before showing how to construct $V$, we present a simple way to make use of $V$ to calculate the rank of a short suffix among $\mathcal{S}(A)$ from its rank among $\mathcal{S}(B)$, and vice versa.

**Lemma 7** *For any short suffix $x$ of $A$, let $r = \mathrm{Rank}(x, \mathcal{S}(A))$ and $r' = \mathrm{Rank}(x, \mathcal{S}(B))$. Then, $r = \mathtt{select}_0(V, r'+1)$ and $r' = \mathtt{rank}_0(V, r)$.*

[4]When $k = \ell - 1$, we assume that $L[\ell]$ has been set to the value of $\Psi_B[0]$. Note that $L[\ell]$ is the rank of $A_\ell$ (or equivalently $B_0$) among $\mathcal{S}(B)$, which is equal to $\mathrm{SA}_B^{-1}[0] = \Psi_B[0]$.

**Proof:** By definition, $V[r] = 0$. In the subarray $V[0..r-1]$, the number of 0's is equal to the number of short suffixes lexicographically smaller than $x$, which is equal to $r'$. Furthermore, $V[r]$ contains the $(r'+1)$-th 0. □

Next, we give the details of constructing $V$. Note that the number of bits in $V$ depends on the size of $A$, which can be as big as $n$.

**Lemma 8** *The bit vector $V$ can be constructed from the array $L$ in $O(n)$ time.*

**Proof:** We assume that $|A|$ bits are allocated for storing $V$ explicitly. We compute $V$ from $L$ as follows: Recall that $L$ stores the ranks of the long suffixes among $\mathcal{S}(B)$. These ranks can solely determine which entries in $V$ store the 1's. We sort the ranks in $L$ in ascending order, denoted as $r_0, r_1, \cdots, r_{\ell-1}$. Then we fill $V$ with the following bits: $r_0$ 0's, a 1, $(r_1 - r_0)$ 0's, a 1, $\cdots$, and finally $(r_{\ell-1} - r_{\ell-2})$ 0's, a 1, followed by all zeroes. □

There are several data structures in the literature that support the rank and select operations on a bit vector in constant time [12, 19]. In particular, we can make use of the recent result by Raman, et al. [19]; precisely, we can build a fully indexable dictionary for $V$ (Lemma 2.3 in [19]) directly from $L$ and we do not need to store the vector $V$ explicitly. The size of this data structure is $\log \binom{n}{\ell} + O(\frac{n \log \log n}{\log n}) = o(n)$ bits, and the construction time remains $O(n)$. With this data structure, the retrieval of $V[i]$ and the queries $\texttt{rank}_0(V,i)$, $\texttt{rank}_1(V,i)$, and $\texttt{select}_0(V,j)$ are performed in $O(1)$ time.

Finally, we are ready to show how to compute $\Psi_A[r]$ for all $r \in [0, m-1]$ where $m = |A|$, the length of $A$. Recall that $\Psi_A[r]$ is defined as $\mathrm{SA}_A^{-1}[\mathrm{SA}_A[r]+1]$, or equivalently, if $A_k$ is the suffix such that $\mathrm{Rank}(A_k, \mathcal{S}(A)) = r$, then $\Psi_A[r] = \mathrm{Rank}(A_{k+1}, \mathcal{S}(A))$. The following two lemmas show how to make use of $V$ to figure out $\Psi_A[r]$ from $\Psi_B[r]$.

**Lemma 9** *Consider any short suffix $A_k$ whose rank among $\mathcal{S}(A)$ is $r$. Then*
- $\mathrm{Rank}(A_{k+1}, \mathcal{S}(B)) = \Psi_B[\texttt{rank}_0(V,r)]$*; and*
- $\mathrm{Rank}(A_{k+1}, \mathcal{S}(A)) = \texttt{select}_0(V, \Psi_B[\texttt{rank}_0(V,r)]+1)$.

**Proof:** Since $A_k$ is a short suffix whose rank among all suffixes of $A$ is $r$, its rank among all suffixes of $B$ is $r' = \texttt{rank}_0(V,r)$. The rank of $A_{k+1}$ among all suffixes of $B$ is $p = \Psi_B[r']$. By Lemma 7, $\Psi_A[r]$, the rank of $A_{k+1}$ among all suffixes of $A$, is $\texttt{select}_0(V, p+1)$. □

**Lemma 10** *Consider any long suffix $A_k$ whose rank among $\mathcal{S}(A)$ is $r$. Then*
- $k = M^{-1}[\texttt{rank}_1(V,r)]$*; and*
- *if* $k < \ell-1$ *then* $\mathrm{Rank}(A_{k+1}, \mathcal{S}(A)) = M[k+1]+L[k+1]$*; otherwise,* $\mathrm{Rank}(A_{k+1}, \mathcal{S}(A)) = \texttt{select}_0(V, \Psi_B[0]+1)$.

**Proof:** Since $x$ is a long suffix, its rank among all long suffixes is $r' = \texttt{rank}_1(V,r)$. By the definition of $M$, $k = M^{-1}[r']$. Note that $k$ is in the range $[0, \ell-1]$. If $k < \ell - 1$, then $\Psi_A[r]$, which is the rank of $A_{k+1}$ among all suffixes of $A$, is equal to $M[k+1] + L[k+1]$ (by Fact 1).

For the special case where $k$ is equal to $\ell - 1$, $\Psi_A[r]$ is equal to the rank of $A_\ell = B_0$ among all suffixes of $A$. We can find this rank as follows: Compute the rank $p$ of $B_0$ among all suffixes of $B$, which is equal to $\mathrm{SA}_B^{-1}[0] = \Psi_B[0]$. Then, by Lemma 7, the rank of $B_0$ among all suffixes of $A$ is $\texttt{select}_0(V, p+1)$. □

Based on the above two lemmas, we can compute $\Psi_A[r]$ sequentially for $r = 0, 1, \ldots, m-1$. For the base case when $r = 0$, we note that $\Psi_A[0]$, which is defined as $\mathrm{SA}^{-1}[0]$ or the rank of $A_0$ among all suffixes of $A$, is exactly $M[0] + L[0]$ (by Fact 1). The details are depicted in Figure 2.

```
Ψ_A[0] ← M[0] + L[0];
for r ← 1 to m − 1
    if V[r] = 0 {    % The suffix with rank r is a short suffix.
        r' ← rank_0(V, r);
        p ← Ψ_B[r'];
        Ψ_A[r] ← select_0(V, p + 1);
    }
    else {    % The suffix with rank r is a long suffix.
        r' ← rank_1(V, r);
        k ← M^{-1}[r'];
        if k < ℓ − 1
            Ψ_A[r] ← M[k + 1] + L[k + 1];
        else {
            p ← Ψ_B[0];
            Ψ_A[r] ← select_0(V, p + 1);
        }
    }
```

Figure 2: Computing $\Psi_A[r]$ sequentially.

Calculating each $\Psi_A[r]$ involves a constant number of $O(1)$ time operations, and the whole procedure takes $O(m) = O(n)$ time. Combining the results of Sections 4.1 and 4.2, we have the following lemma.

**Lemma 11** *Suppose that $\Psi_B$ is given. Computing all the auxiliary data structures ($M$, $M^{-1}$, $L$, and $V$) and then enumerating the values of $\Psi_A$ can be done in $O(\ell \log n + n)$ time. Excluding the space for representing $\Psi_A$ and $\Psi_B$, the working space required is $4\ell\lceil\log n\rceil + n + o(n)$ bits.*

As mentioned in Section 2, we can construct a compact representation for $\Psi_A$ using $O(n(H_0 + 1))$ bits. For its auxiliary data structures, the Raman et al.'s dictionary for the $\alpha(c)$ values can be computed directly in $O(\ell + |\Sigma|)$ time based on examining $M^{-1}$ sequentially and the corresponding dictionary in $\Psi_B$ (i.e., the one for the $\alpha_B(c)$ values), while the remaining two data structures are computed along with the construction of the compact representation of $\Psi_A$, using an extra $O(n)$ time.

Together with Lemma 11, we conclude this section with the following result.

**Theorem 1** *Given a string $T$ of length $n$, the $\Psi$ array of $T$ can be computed in $O(n \log n)$ time using $O(n(H_0 + 1))$ bits.*

**Proof:** The construction is divided into $\lceil n/\ell \rceil = O(\log n)$ phases. Recall that $\ell = \Theta(n/\log n)$. Each phase takes $O(\ell \log n + n) = O(n)$ time, and the overall time is $O(n \log n)$.

For the space requirement, it takes $4\ell\lceil \log n \rceil + o(n)$ bits in addition to that for two $\Psi$ arrays and their auxiliary data structures. The total space is thus $O(n(H_0+1)) + 4\ell\lceil \log n \rceil$ bits. Since $\ell = \Theta(n/\log n)$, the theorem follows. □

# 5 Constructing Other Indexes

We have shown an algorithm to construct the array $\Psi$, which is the basic form of the CSA, using $O(n(H_0+1))$ bits working space. Here, we show how to apply the algorithm to construct the hierarchical CSA, and how to convert $\Psi$ into the FM-index in a space-efficient manner.

## 5.1 Constructing the hierarchical CSA structures

The original compressed suffix array [7] is a hierarchical data structure which supports efficient retrieval of any SA value in $O(\log \log_{|\Sigma|} n)$ time. Let $k$ be any integer in the range $[0, \log \log_{|\Sigma|} n]$. Let $T_k$ denote the string obtained by concatenating every $2^k$ characters of $T$. The string $T_k$ can be viewed as a text whose characters are drawn from $\Sigma^{2^k}$. The hierarchical CSA of $T$ consists of the $\Psi_k$ functions built on top of $T_k$, where $k = 0, 1, \ldots, \log \log_{|\Sigma|} n$. And, at the final level ($k = \log \log_{|\Sigma|} n$), it stores explicitly the $\mathrm{SA}_k$ values for the corresponding $T_k$. Each $\Psi_k$ function is coupled with a bit-vector $B_k$ and the Jacobson's data structure for $B_k$ so that the rank function $rank(B_k, i)$—which returns the number of 1's in $B_k[0..i]$—can be answered in $O(1)$ time. In summary, the total space to store the hierarchical CSA is at most $O(n(H_0 \log \log n + 1))$ bits. SA[$i$] can be computed recursively in $O(\log \log_{|\Sigma|} n)$ time as follows:

$$\mathrm{SA}_k[i] = \begin{cases} 2 \cdot \mathrm{SA}_{k+1}[rank(B_k, i)] & \text{if } B_k[i] = \mathtt{1} \\ \mathrm{SA}_k[\Psi_k(i)] - 1 & \text{if } B_k[i] = \mathtt{0} \end{cases}$$

For the construction, $\Psi_k$ can be computed in $O((n \log n)/2^k)$ time based on Theorem 1. After that, by letting $t = \mathrm{SA}_k^{-1}[0]$ and computing $\Psi_k^i[t]$ iteratively for each $i$, we obtain the vector $B_k$ and its auxiliary data structure in $O(n/2^k)$ time. For the $\mathrm{SA}_k$ at the final level, it can be computed in $O(n)$ time, since $T_k$ is a string of $O(n/\log n)$ characters. Thus, the total time is $O(n \log n)$. For the space requirement, apart from the space of the final output, the above algorithm takes an extra $O(n(H_0 + 1))$ bits for working space. This gives the following theorem.

**Theorem 2** *Given the text $T$ over an alphabet $\Sigma$, the hierarchical structure of CSA in [7] can be computed in $O(n \log n)$ time and $O(n(H_0 + 1))$ bits of working space in addition to the output, where $H_0$ denotes the order-0 entropy of $T$. With this data structure, each* SA *value can be reported in $O(\log \log_{|\Sigma|} n)$ time.*

## 5.2 Converting $\Psi$ into the FM-index

Apart from CSA, there is another compressed index for suffix array called FM-index [3], 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 $T$ [17]. 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 $\Psi$ array of $T$, we observe that for any $p$, $\mathrm{SA}[\Psi^k[p]] = \mathrm{SA}[p] + k$ [21]. 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 $\mathrm{SA}[\Psi^k[p]] = k$. Immediately, we can set $W[\Psi^k[p]] = T[k-1]$. Since each computation of $\Psi$ takes $O(1)$ time, $W$ can be constructed in $O(n)$ time.

Thus, we have the following theorem.

**Lemma 12** *Given the text $T$ and the $\Psi$ array 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, FM-index can be created by encoding the transformed text $W$ using Move-to-Front encoding and Run-Length encoding [3]. When the alphabet size is small, precisely, when $|\Sigma| \log |\Sigma| = O(\log n)$, Move-to-Front encoding and Run-Length encoding can be done in $O(n)$ time based on a pre-computed table of $o(n)$ bits. In summary, this encoding procedure takes $O(n)$ time using $o(n)$-bit space in addition to the output index. Thus, we have the following result.

**Theorem 3** *Given the text $T$ over a small alphabet $\Sigma$ such that $|\Sigma| \log |\Sigma| = O(\log n)$, and the $\Psi$ function of $T$, we can construct the FM-index of $T$ in $O(n)$ time using $O(n \log |\Sigma|)$ bits in addition to the output index.*

# References


[1] M. Burrows and D. J. Wheeler. A Block-sorting Lossless Data Compression Algorithm. Technical Report 124, Digital Equipment Corporation, Paolo Alto, CA, USA, 1994.

[2] D. R. Clark and J. I. Munro. Efficient Suffix Trees on Secondary Storage. In *Proceedings of Symposium on Discrete Algorithms*, pages 383–391, 1996.

[3] P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. In *Proceedings of Symposium on Foundations of Computer Science*, pages 390–398, 2000.

[4] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proceedings of Symposium on Discrete Algorithms*, pages 269–278, 2001.

[5] R. Grossi, A. Gupta, and J. S. Vitter. High-Order Entropy-Compressed Text Indexes. In *Proceedings of Symposium on Discrete Algorithms*, pages 841–850, 2003.

[6] R. Grossi, A. Gupta, and J. S. Vitter. When Indexing Equals Compression: Experiments with Compressing Suffix Arrays and Applications. In *Proceedings of Symposium on Discrete Algorithms*, pages 636–645, 2004.

[7] R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. In *Proceedings of 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. *SIAM Journal on Computing*, to appear.

[9] W. K. Hon, T. W. Lam, W. K. Sung, W. L. Tse, C. K. Wong, and S. M. Yiu. Practical Aspects of Compressed Suffix Arrays and FM-index in Searching DNA Sequences. In *Proceedings of Workshop on Algorithm Engineering and Experiments*, 2004. to appear.

[10] W. K. Hon, K. Sadakane, and W. K. Sung. Breaking a Time-and-Sapce Barrier in Constructing Full-Text Indices. In *Proceedings of Symposium on Foundations of Computer Science*, pages 251–260, 2003.

[11] E. Hunt, M. P. Atkinson, and R. W. Irving. A database index to large biological sequences. In *Proceedings of International Conference on Very Large Data Bases*, pages 410–421, 2000.

[12] G. Jacobson. Space-efficient Static Trees and Graphs. In *Proceedings of Symposium on Foundations of Computer Science*, pages 549–554, 1989.

[13] S. Kurtz. Reducing the Space Requirement of Suffix Trees. *Software Practice and Experiences*, 29:1149–1171, 1999.

[14] 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 International Conference on Computing and Combinatorics*, pages 401–410, 2002.

[15] N. J. Larsson and K. Sadakane. Faster Suffix Sorting. Technical Report LU-CS-TR:99-214, Lund University, 1999.

[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] G. Manzini. An Analysis of the Burrows-Wheeler Transform. *Journal of the ACM*, 48(3):407–430, 2001.

[18] E. M. McCreight. A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[19] R. Raman, V. Raman, and S. S. Rao. Succinct Indexable Dictionaries with Applications to Encoding $k$-ary Trees and Multisets. In *Proceedings of Symposium on Discrete Algorithms*, pages 233–242, 2002.

[20] R. F. Rice. Some practical universal noiseless coding techniques. Technical Report JPL-79-22, Jet Propulsion Laboratory, Pasadena, CA, USA, 1979.

[21] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003. A preliminary version appears in ISAAC 2000.

[22] K. Sadakane and T. Shibuya. Indexing Huge Genome Sequences for Solving Various Problems. In *Genome Informatics*, pages 175–183, 2001.

[23] J. Seward. The `bzip2` and `libbzip2` official home page, 1996. `http://sources.redhat.com/bzip2/`.