Breaking a Time-and-Space Barrier in Constructing Full-Text Indices

Wing-Kai Hon*
The University of Hong Kong
wkhon@csis.hku.hk

Kunihiko Sadakane†
Kyushu University
sada@csce.kyushu-u.ac.jp

Wing-Kin Sung‡
National University of Singapore
ksung@comp.nus.edu.sg

Abstract

Suffix trees and suffix arrays are the most prominent full-text indices, and their construction algorithms are well studied. It has been open for a long time whether these indices can be constructed in both $o(n \log n)$ time and $o(n \log n)$-bit working space, where n denotes the length of the text. In the literature, the fastest algorithm runs in $O(n)$ time, while it requires $O(n \log n)$-bit working space. On the other hand, the most space-efficient algorithm requires $O(n)$-bit working space while it runs in $O(n \log n)$ time.

This paper breaks the long-standing time-and-space barrier under the unit-cost word RAM. We give an algorithm for constructing the suffix array which takes $O(n)$ time and $O(n)$-bit working space, for texts with constant-size alphabets. Note that both the time and the space bounds are optimal. For constructing the suffix tree, our algorithm requires $O(n \log^{\epsilon} n)$ time and $O(n)$-bit working space for any $0 < \epsilon < 1$. Apart from that, our algorithm can also be adopted to build other existing full-text indices, such as Compressed Suffix Tree, Compressed Suffix Arrays and FM-index.

We also study the general case where the size of the alphabet $\mathcal{A}$ is not constant. Our algorithm can construct a suffix array and a suffix tree using optimal $O(n \log |\mathcal{A}|)$-bit working space while running in $O(n \log \log |\mathcal{A}|)$ time and $O(n \log^{\epsilon} n)$ time, respectively. These are the first algorithms that achieve $o(n \log n)$ time with optimal working space, under a reasonable assumption that $\log |\mathcal{A}| = o(\log n)$.

* Department of Computer Science and Infomation Systems, The University of Hong Kong, Hong Kong.

† Department of Computer Science and Communication Engineering, Kyushu University, Japan.

‡ School of Computing, National University of Singapore, Singapore.

1. Introduction

Due to the advance in information technology and biotechnology, the amount of text data is increasing exponentially. To assist users to locate their required information, the role of indexing data structures has become more and more important. For texts with word boundary such as English, inverted index [8] is used since it enables fast queries and is space-efficient. However, for texts without word boundary like DNA/protein sequences or Chinese/Japanese texts, inverted index is not suitable. In this case, we need full-text indices, that is, indexing data structures which make no assumption on the word boundary. Suffix trees [20] and suffix arrays [19] are two fundamental full-text indices in the literature, which find numerous applications in areas including data mining [27] and biological research [9]. For the rest of the full-text indices, almost all of them are originated from these two data structures.

Suffix trees and suffix arrays are very useful since they allow us to perform pattern searching efficiently. Consider a text with n characters. Given the suffix tree, we can search for a pattern P within the text using $O(|P|)$ time, which is independent of the text size. For suffix array, the searching time is $O(|P| + \log n)$,[1] which is only a bit slower. One more advantage of suffix array is that even if this indexing structure is placed in external memory, it still can achieve good I/O performance for searching [3]. Despite their efficiency in searching, suffix trees and suffix arrays cannot be built easily when n is large. The construction algorithms for both of them are either too slow, or require too much working space.

For instance, when we optimize the construction time, based on the work from Weiner [29], McCreight [20], Ukkonen [28], and Farach [4], a suffix tree and a suffix array can be built in $O(n)$ time. However, the working space required is $\Omega(n \log n)$ bits.

1 We use the notation $\log_b^c n = (\log n / \log b)^c$ to denote the c-th power of the base-b logarithm of n. Unless specified, we use $b = 2$.

On the other hand, when we optimize the construction working space, based on the recent work by Lam et al. [18], we can first build the Compressed Suffix Array (CSA) of [6] and then convert it to the suffix tree and the suffix array. Although such approach reduces the working space to $O(n)$ bits, the execution time is increased to $O(n \log n)$. Another solution is to rely on external memory [3] to control the working space. However, the time complexity is even worse, not to mention the increase in I/O burden.

It has been open for a long time whether the suffix tree and the suffix array can be constructed in $o(n \log n)$ time and $o(n \log n)$-bit working space. The need to break this time-and-space barrier is illustrated in a concrete example that arises in practice. Suppose we would like to construct a suffix array for human genome (of length approximately 3 billion). The fastest known algorithm runs in linear time. However, it requires 40 Gigabytes working space [17]. Such memory requirement far exceeds the capacity of ordinary computers. On the other hand, if we apply the most space-efficient algorithm, the working space required is roughly 3 Gigabytes, which is possible to be implemented on a PC nowadays. The time required, however, is more than 20 hours [18], which is a bit slow.

Apart from suffix trees and suffix arrays, we observe that the other full-text indices also suffer from the same time-and-space barrier during the construction phase. Such barrier may prevent these indices to become useful for large-scale applications.[2] Table 1 summarizes the performance of the best known algorithms for constructing these full-text indices.

index	algorithm	time	space (bits)
SA CSA FM	opt time [19]	$O(n)$	$O(n \log n)$
	opt space [18]	$O(n \log n)$	$O(n)$
	this paper	$O(n)$	$O(n)$
ST CST	opt time [4]	$O(n)$	$O(n \log n)$
	opt space [12]	$O(n \log n)$	$O(n)$
	this paper	$O(n \log^{\epsilon} n)$	$O(n)$

The acronym ST, SA, CST, CSA, FM represent suffix tree, suffix array, compressed suffix tree, compressed suffix array, and FM-index, respectively. Note that $0 < \epsilon < 1$.

Table 1. Construction times for full-text indices.

1.1. Our Results

Our results are based on the following model. Firstly, we assume a unit-cost RAM with word size $O(\log U)$ bits, where $n \leq U$, in which standard arithmetic and bitwise boolean operations on word-sized operands can be performed in constant time [1, 10]. For counting the working space, we do not include the space for the output of the full-text indices, and we are allowed to write the output once.[3] Under such computation model, this paper solves the problem by giving the following algorithms.

1. An algorithm which constructs the suffix array in $O(n)$ time and $O(n)$-bit working space;
2. An algorithm which constructs the suffix tree in $O(n \log^{\epsilon} n)$ time and $O(n)$-bit working space for $0 < \epsilon < 1$.

As far as the authors know, they are the first known algorithms which run in $o(n \log n)$ time and $o(n \log n)$-bit working space.

Besides, our algorithms can actually be adopted to build other full-text indices, including CSA, Compressed Suffix Tree (CST) [12], and FM-index [5]. The performance of our algorithms for constructing these indices are summarized in Table 1. Another application of our algorithm is that, it can act as a time and space efficient data compression algorithm for the block sorting [2], which is widely used in `bzip2` [26].

We also study the general case where the alphabet size is not constant. Let $\mathcal{A}$ be the alphabet, and $|\mathcal{A}|$ denote its size. Our algorithm can construct the suffix array and the suffix tree using $O(n \log |\mathcal{A}|)$-bit working space, while runs in $O(n \log \log |\mathcal{A}|)$ time and $O(n \log^{\epsilon} n)$ time, respectively, for any fixed ϵ with $0 < \epsilon < 1$. These are the first algorithms that achieve $o(n \log n)$ time with optimal working space, under a reasonable assumption that $\log |\mathcal{A}| = o(\log n)$.

1.2. The Main Techniques

To achieve small working space, we make use of the Ψ function [6] of CSA and the Burrows-Wheeler (BW) text [2] as our tools, where they can act as an implicit representation of any suffix tree generated during the process of construction. Both the Ψ function and the BW text can be stored in $O(n)$-bit space. Moreover, given them, we can construct the suffix tree and the suffix array in $O(n \log^{\epsilon} n)$ time and $O(n)$ time, respectively for $0 < \epsilon < 1$.

The other reason for choosing these two data structures as our tools is that, they possess different strengths, which complement nicely the weaknesses of the other. For the Ψ function, it allows efficient pattern query, while it is difficult to update the function in response to the change in the underlying suffix tree. On the other hand, the BW text allows easier and faster update, while it does not support efficient pattern query. In our construction algorithm, efficient

2 Zobel et al. [31] and Crauser and Ferragina [3] both mentioned the importance of construction algorithms to the usefulness of the index.

3 Based on this model, the output can be written directly to the secondary storage, without occupying the main memory.

queries and fast updates are frequently required, so that both data structures are alternately used in order to utilize their advantages.

Another finding that leads to the improvement is related to the backward search algorithm, which is used to find a pattern within the text based on the Ψ function. If we apply a known method [25], given the Ψ function for the text, each step of the algorithm requires $O(\log n)$ time in general. This paper presents an $O(n)$-bit auxiliary data structure which supports each backward search step in $O(\log\log|\mathcal{A}|)$ time instead. As our construction algorithm executes the backward search frequently, the overall running time is thus sped up because of this improvement.

Finally, the framework of our algorithm is borrowed from Farach's linear time suffix tree construction algorithm [4] which first constructs two suffix trees, one for odd-position suffixes and one for even-position suffixes, and then merge the two trees together. However, the actual implementation is different, as we have carefully avoided to store suffix pointers explicitly and thus saving $O(n\log n)$ bits in total.

The remaining of this paper is organized as follows. Section 2 is a preliminary section which introduces some useful tools. Section 3 shows the improved backward search algorithm that we base on. Section 4 describes the framework for the construction algorithm, while Sections 5 and 6 detail the main steps of the algorithm. Finally, we give a conclusion in Section 7.

$S = $ `a b b a a a b $`

i	$C[i]$	$SA[i]$	$\Psi[i]$	
1	b	8	5	$
2	b	4	3	a a a b $
3	a	5	4	a a b $
4	a	6	6	a b $
5	$	1	8	a b b a a a b $
6	a	7	1	b $
7	b	3	2	b a a a b $
8	a	2	7	b b a a a b $

(a)

$S_o = $ `ab ba aa b$`

i	$C_o[i]$	$SA_o[i]$	$\Psi_{S_o}[i]$	
1	ba	3	3	aa b$
2	b$	1	4	ab ba aa b$
3	aa	4	2	b$
4	ab	2	1	ba aa b$

(b)

$S_e = $ `bb aa ab $a`

i	$C_e[i]$	$SA_e[i]$	$\Psi_{S_e}[i]$	
1	ab	4	4	$a
2	bb	2	3	aa ab $a
3	aa	3	1	ab $a
4	$a	1	2	bb aa ab $a

(c)

Figure 1. Suffix arrays, the Ψ functions and the C arrays of (a) S, (b) S_o, and (c) S_e.

2. Preliminaries

This section is divided into two parts. The first part gives some basic definitions, including the Ψ function of the Compressed Suffix Array (CSA) and the output array C of the Burrows-Wheeler Transformation (BWT), which are the main tools for our algorithm. Then we discuss the duality between the Ψ function and the C array.

2.1. Basic Definitions

Let $T[1..n] = T[1]T[2]\cdots T[n]$ be a string of length n over an alphabet $\mathcal{A}$, with $|\mathcal{A}| \le n$. We assume that $T[n] = \$$ is a unique terminator which is alphabetically smaller than other symbols. For any $i = 1, \ldots, n$, $T[i..n]$ is a suffix of T. The suffix array $SA[1..n]$ of T is an array of integers such that $T[SA[i]..n]$ is lexicographically the i-th smallest suffix of T.

The Ψ_T function, or simply Ψ, is the main component of the CSA [6]. It is defined as

$$\Psi[i] = SA^{-1}[SA[i] + 1].$$

See Figure 1(a) for an example.

The Ψ function can be encoded in $O(n\log|\mathcal{A}|)$ bits [7]. An encoding scheme of Ψ will be presented shortly, which will be used in our algorithm. Before describing the details, we need one observation.

Observation 1 *For the suffix array and the Ψ function of the text T,*

- *Characters $T[SA[i]]$ ($i = 1, 2, \ldots, n$) are alphabetically sorted.*
- *$\Psi[i] < \Psi[j]$ if $i < j$ and $T[SA[i]] = T[SA[j]]$.*

Therefore Ψ function is piece-wise increasing. To make it an increasing sequence, we make use of a function $\rho(c, x) = cn + x - 1$, and transform the Ψ function into $\Psi'[i] = \rho(T[SA[i]], \Psi[i])$.

Lemma 1 [7, 23] *The Ψ' function can be encoded in $O(n\log|\mathcal{A}|)$ bits, so that each $\Psi'[i]$ can be retrieved in constant time.*

Proof: We divide each $\Psi'[i]$ into two parts q_i and r_i, where q_i is the first $\log n$ bits and r_i is the remaining $\log|\mathcal{A}|$ bits. We use the difference values $q_1, q_2 - q_1, \ldots, q_n - q_{n-1}$ to represent the q_i's. Note that all the difference values are non-negative, so that they can be encoded in a bit-vector

B_1 using unary codes.[4] The encoding would have exactly n 1's where the i-th 1 is at position $i + q_i$. Also, the total number of 0's is q_n which is at most n. Thus B_1 uses $2n$ bits. The r_i's are stored explicitly in an array $B_2[1..n]$ where each entry occupies $\log|\mathcal{A}|$ bits. Thus B_2 occupies $n\log|\mathcal{A}|$ bits. Moreover, an auxiliary data structure of $O(n/\log\log n)$ bits is constructed in $O(n)$ time to enable the retrieval of any q_i in constant time [22]. Then the total size is $n(2 + \log|\mathcal{A}|) + O(n/\log\log n)$.

Since q_i and r_i can be computed in constant time, so can $\Psi'[i] = |\mathcal{A}|\, q_i + r_i$. □

Corollary 1 *The Ψ function can be encoded as Ψ' in $O(n\log|\mathcal{A}|)$ bits, so that each $\Psi[i]$ can be retrieved in constant time.*

Proof: The retrieval time follows since $\Psi[i] = (\Psi'[i] \bmod n) + 1$. □

The BWT [2] is used extensively in many compression algorithms. It converts the string $T[1..n]$ into the $C[1..n]$ array, which 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(a) for an example.

To conclude this part, we state a few results that construct some popular full-text indices given the Ψ function and the C array of T.

Lemma 2 *Given the text T and the Ψ function, we can construct the suffix array and the compressed suffix array in $O(n)$ time and $O(n\log|\mathcal{A}|)$-bit working space.*

Lemma 3 [12] *Given the text T and the Ψ function, we can construct the suffix tree and the compressed suffix tree in $O(n\log^{\epsilon} n)$ time and $O(n\log|\mathcal{A}|)$-bit working space, for any fixed ϵ with $0 < \epsilon < 1$.*

Lemma 4 *Given the C array of T, we can construct the FM-index in $O(n)$ time and $O(n\log|\mathcal{A}|)$-bit working space, for $|\mathcal{A}|\log|\mathcal{A}| = O(\log n)$.*[5]

2.2. Duality between the Ψ function and the C array

This section describes the relation between the Ψ function and the C array of T. We begin with giving an intuitive meaning to the C array. For a suffix $T[i..n]$, we refer the character $T[i-1]$ (or $T[n]$ if $i = 1$) to be its *preceding* character.

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

5 In the original paper [5], the alphabet size $|\mathcal{A}|$ is assumed to be constant.

1. Compute $L_1[c]$ to store the number of occurrences of each $c \in \mathcal{A}$.
2. Compute $L_2[c]$ to store the cumulative frequency of the characters, that is, $L_2[c] = \sum_{d<c} L_1[c]$ for $c \in \mathcal{A}$.
3. Let $L_3[c]$ be an array that stores the number of characters c seen so far.
4. Initialize L_3 to be 0.
5. For $i = 1, 2, \ldots, n$
   Let $c = C[i]$, $k = L_2[c] + L_3[c]$
   Compute $x = \rho(c, i)$
   Let $q = x$ div $|\mathcal{A}|$, $r = x \bmod |\mathcal{A}|$
   Set $B_1[k + q] = 1$ and $B_2[k] = r$
   Increment $L_3[c]$ by one

Figure 2. Computing Ψ from C.

Observation 2 [2] *The $C[1..n]$ obtained from BWT of T is an array of characters in $\mathcal{A}$ such that $C[i]$ is the preceding character of $T[SA[i]..n]$. We refer the C array as the* compact representation *of Ψ.*

It is known that Ψ and C are one-to-one corresponding. We show that the transformation between them can be done in linear time and in $O(n\log|\mathcal{A}|)$-bit space. Firstly, we give a property about the C array and the Ψ function.

Definition 1 *Given an array of characters $x[1..n]$, we define the* stable sorting order *of $x[i]$ in x to be the number of characters in x which is alphabetically smaller than $x[i]$, plus the number of characters $x[j]$ with $j \leq i$ which is equal to $x[i]$. This is in fact the order of $x[i]$ after stable sorting.*

Lemma 5 [2] *Let k be the stable sorting order of $C[i]$ in C. Then $\Psi[k] = i$.*

Proof: Let Y_i denote the string $C[i]T[SA[i]..n]$. Observe that Y_i is a suffix of T in general, and intuitively the set $Y = \{Y_j | 1 \leq j \leq n\}$ is the same as the set of suffixes of T.[6] Now, suppose that Y_i is the k'-th smallest string in Y. We have $Y_i = T[SA[k']..n]$. By omitting the first character on both sides, we get $T[SA[i]..n] = T[SA[k'] + 1..n]$, which implies $i = SA^{-1}[SA[k'] + 1] = \Psi[k']$.

It remains to prove $k' = k$. Note that excluding the first characters of Y_i's, they are in the form $T[SA[i]..n]$ which are lexicographically sorted. Therefore, the lexicographic

6 Suppose $suf(T)$ is the set of suffixes of T. Then $suf(T)$ and Y differ only by the string $T[n] \in suf(T)$ and $T[n]T[1..n] \in Y$. These two exception strings have the same rank among the corresponding set (as they begin with the same first character, which is unique among the first characters of other strings), and the rank of the other strings are identical in both sets. Thus the two sets are 'intuitively' the same.

order of Y_i is equal to the stable sorting order of $C[i]$, which is k. □

The next two lemmas show the linear time conversion between $C[1..n]$ and $\Psi[1..n]$.

Lemma 6 *Given $C[1..n]$, we can store $\Psi[1..n]$ in $O(n)$ time and $O(n \log |\mathcal{A}|)$ bits.*

Proof: We construct Ψ' from C as an encoding of Ψ (Corollary 1). To construct Ψ', we create a bit-vector $B_1[1..2n]$ and initialize it to 0. We also create an array $B_2[1..n]$ where each entry occupies $\log |\mathcal{A}|$ bits. These are used to store Ψ' using the scheme in Lemma 1. We iteratively compute the stable sorting order k for $C[i]$. By Lemma 5, $\Psi[k] = i$. Thus, we store $\Psi'[k]$ with $x = \rho(T[SA[k]], \Psi[k]) = \rho(C[i], i)$. By the proof of Lemma 1, x is divided into two parts q and r, where $q = x \operatorname{div} |\mathcal{A}|$ is the first $\log n$ bits, and $r = x \bmod |\mathcal{A}|$ is the remaining bits. For q, a 1 is stored at $B_1[k+q]$. For r, it is stored at $B_2[k]$. Thus, we have the algorithm shown in Figure 2.

The total space for B_1 and B_2 is $O(n \log |\mathcal{A}|)$ bits. The auxiliary data structure for Ψ is $O(n/\log\log n)$ bits [22]. It is obvious that the above algorithm runs in $O(n + |\mathcal{A}|) = O(n)$ time. Note that L_1, L_2, L_3 each occupies $|\mathcal{A}| \log n \le n \log |\mathcal{A}|$ bits because $|\mathcal{A}| \le n$. Thus we use $O(n \log |\mathcal{A}|)$-bit temporary space. □

Lemma 7 *Given $\Psi[1..n]$ and T, we can construct $C[1..n]$ in $O(n)$ time and $O(n \log |\mathcal{A}|)$-bit space.*

Proof: Let $t = SA^{-1}[n]$. We claim that $SA[\Psi^k[t]] = k$ under modulo-n arithmetic. By definition, $SA[\Psi[i]] = SA[i] + 1$. Thus, $SA[\Psi^k[t]] = SA[\Psi^{k-1}[t]] + 1 = \cdots = SA[\Psi[t]] + k - 1 = SA[t] + k = n + k = k$.

Hence, we have $C[\Psi^k[t]] = T[k-1]$. To construct C, we first compute t. Since $T[n]$ is unique, t is in fact equal to 1 plus the number of characters $T[i]$ which is smaller than $T[n]$. We obtain t by scanning T once in $O(n)$ time. Then, we iteratively compute $\Psi^i[t]$ and set $C[\Psi^i[t]] = T[i-1]$, for $i = 1$ to n. The total time required is $O(n)$, and the space is $O(n \log |\mathcal{A}|)$ bits for C, T and Ψ. □

3. Improved Backward Search Algorithm

This section gives some lemmas for a searching algorithm based on the Ψ function, which is known as backward search. First, we introduce a definition.

Definition 2 *Let P be a string and X be a set of strings. Define* rank *of P among X to be the number of strings in X which is lexicographically not greater than P.*

Given the Ψ function of a text S over an alphabet Δ, we can augment it with a data structure of size $O(|S| \log |\Delta|)$, so that it can support the backward search algorithm of [24]. The above data structure can be computed from S in $O(|S|)$ time. With this *augmented*-Ψ (i.e., the Ψ function and its augmented data structure), we can apply the backward search algorithm to determine the rank of any pattern $P[1..\ell]$ among the suffixes of the string S efficiently. The algorithm is analogous to that of the FM-index [5]. Basically, it achieves the goal by computing the rank of $P[i..\ell]$ based on the rank of $P[i+1..\ell]$, for $i = \ell - 1$ to 1, as shown in the following lemma. Recall that $\rho(c, x) = cm + x - 1$ and $\Psi'[i] = \rho(S[SA[i]], \Psi[i])$ so that Ψ' is monotone increasing.

Lemma 8 *Given the augmented-Ψ function of S, for any string $P[1..\ell]$, the rank of P among the suffixes of S can be found by the backward search algorithm in ℓ steps. In particular, at step i, the rank of $P[\ell - i + 1..\ell]$ is determined.*

Proof: Let r_{i+1} be the rank of $P[i+1..\ell]$ among suffixes of S. The lemma follows since the rank r_i of $P[i..\ell]$ can be computed based on r_{i+1}, which is equal to $j \in [1, m]$ with $\Psi'[j] \le \rho(P[i], r_{i+1}) < \Psi'[j+1]$. □

We refer each step in the algorithm of Lemma 8 a *backward search step*. In the original paper [24], each backward search step takes $O(\log |S|)$ time based on the augmented-Ψ. In the remaining of the section, we show the time required by each such step can be improved to $O(\log \log |\Delta|)$.

Let $pred(Q, x)$ denote the rank of the predecessor of x in a set Q. Then, r_i can be computed by:

$$r_i \equiv pred(\Psi', \rho(P[i], r_{i+1}))$$

As a sentinel we define $\Psi'[-1] = -1$ and $\Psi'[m|\Delta|] = m|\Delta|$, where Δ denotes the alphabet of S.

The following theorem is useful for the computation of the function *pred*. The proof of which will be defered to Section 3.1.

Theorem 1 *Given n distinct w-bit numbers in increasing order, where $w = \Theta(\log n)$, a data structure of size $O(nw)$ bits supporting $O(\log w)$-time predecessor queries can be constructed in $O(n)$ time and $O(nw)$-bit working space.*

Based on Theorem 1, we have the following lemma.

Lemma 9 *Let S be a string of length m over an alphabet Δ. An auxiliary data structure for the augmented-Ψ function of S of size $O(m + |\Delta|)$ bits can be constructed in $O(m + |\Delta|)$ time, which supports each backward search step in $O(\log \log |\Delta|)$ time.*

Proof: Recall that the Ψ function (or Ψ' function) is stored in $m(2 + \log |\Delta|)$ bits where the first $\log m$ bits are encoded by unary codes in a bit-vector B_1 of size at most $2m$ bits, and the remaining $\log |\Delta|$ bits are encoded in an array B_2 as it is.

To prove the lemma, we show a data structure that for every i, r_i can be computed from $P[i]$ and r_{i+1} in $O(\log\log|\Delta|)$ time. Let $x = \rho(P[i], r_{i+1})$. Then r_i, which is $pred(\Psi', x)$, is in fact the rank of the predecessor of x in Ψ'. If the first $\log m$ bits of the predecessor x is different from the first $\log m$ bits of x (i.e., either the first $\log m$ bits of x is distinct from the $\log m$ bits of all Ψ' values, or x is smaller than the smallest Ψ' value with the same $\log m$ bits), then the predecessor of x and its rank can be found immediately by constant number of *rank* and *select* queries on B_1, which takes constant time [13, 22].

Otherwise, B_1 reveals a group of Ψ' values which share the same first $\log m$ bits with x, and the predecessor of x must be within the group. If the size of the group is smaller than $\log|\Delta|$, the predecessor of x can be found by a binary search in $O(\log\log|\Delta|)$ time. On the other hand, if the size of the group is larger than $\log|\Delta|$, we make use of the following data structure to facilitates predecessor query:

For all the groups (of Ψ' values) whose size is greater than $\log|\Delta|$, we divide each of them into smaller *units* of size $\Theta(\log|\Delta|)$, and select the smallest element from each unit as a representative. The representatives are maintained as partitions of size $O(|\Delta|/\log|\Delta|)$. Precisely, in each partition, a representative is labeled by its last $\log|\Delta|$ bits, with a group-tag of $\log|\Delta|$ bits added to the front, so that the tag preserves the ordering of the representatives, and the representatives from the same group receive the same tag. The tagged representatives are then stored in a data structure $\mathcal{D}$ of Theorem 1.

To find the predecessor of x (when it is within a 'large' group, say, G), we get the last $\log|\Delta|$ bits of x, and add the group-tag of G to the front of it. The unit (say, U) containing the predecessor of x can be found from querying $\mathcal{D}$ (Precisely, we locate the representative of U from $\mathcal{D}$). Afterwards, we can search for the predecessor of x among the elements in U. The total time required is $O(\log\log|\Delta|)$.

The space occupancy is as follows. There are at most $m/\log|\Delta|$ representatives, so that the data structures of Theorem 1 in total occupy $O(m)$ bits. For storing the pointers (such as from each group to the first and last element in the group, or from each group to the corresponding data structure of Theorem 1), it can be done by using constant number of $O(m + |\Delta|)$-bit bit-vectors with constant time *rank* and *select* supported. The total space required is $O(m + |\Delta|)$ bits.

For the construction time, the data structure of Theorem 1 takes at most $O(m + |\Delta|)$ time, and all the auxiliary data structures supporting constant time *rank* and *select* query in total requires $O(m + |\Delta|)$ time. Thus, the construction time follows, and this completes the proof of the lemma. □

3.1. Data Structure for Efficient Predecessor Query

This section is devoted to proving Theorem 1. We begin with two supporting lemmas. The first one is on perfect hash function, which is obtained by rephrasing the result of Section 4 of [11] as follows.

Lemma 10 *Given x b-bit numbers, where $b = \Theta(\log x)$, a data structure of size $O(xb)$ bits supporting $O(1)$-time existential query can be constructed in $O(x \log x)$ time and $O(xb)$-bit working space.*

The second one is derived from adapting a result in [21, 30] based on Lemma 10.

Lemma 11 *Given z w-bit numbers, where $w = \Theta(\log z)$, a data structure of size $O(zw^2)$ bits supporting $O(\log w)$-time predecessor queries can be constructed in $O(zw \log(zw))$ time and $O(zw^2)$-bit working space.*

Proof: It is shown that predecessor queries can be solved in $O(\log w)$ time, if existential query for all prefixes of the z numbers can be answered in $O(1)$ time [21, 30]. The idea is that, given a w-bit number k, its longest common prefix with the z numbers can be found by binary search (on the length) using $O(\log w)$ existential queries, and such a prefix uniquely determines the predecessor of k.

Notice that only $O(zw)$ strings can be a prefix of the z numbers and each can be represented in $\Theta(w)$ bits. Applying Lemma 10 on this set of strings (with $x = O(zw)$ and $b = \Theta(w)$), we have the required data structure. The lemma thus follows. □

Now, we prove Theorem 1 with the data structure constructed as follows.

1. Let $k_0 < k_1 < \cdots < k_{n-1}$ be n numbers stored in ascending order by an array.
2. Partition the n numbers into n/w^2 lists, each containing w^2 numbers. Precisely, the lists are in the form $\{k_i, k_{i+1}, \ldots, k_{i+w^2-1}\}$, where $i \equiv 0 \pmod{w^2}$.
3. Let the smallest element in each list be its representative. Construct a data structure for predecessor query for these representatives based on Lemma 11.

The above data structure occupies $O(nw)$ bits, and can be constructed in $O(n)$ time and $O(nw)$-bit working space. With such a data structure, the predecessor of x among the n numbers can be found in $O(\log w)$ time as follows.

1. Find the predecessor of x among the n/w^2 representatives of the lists. Let this element be r.
2. The predecessor of x among the n numbers must now lie in the list represented by r. Binary search on the w^2 elements in the list to locate the predecessor.

Both steps thus take $O(\log w)$ time. This completes the proof of Theorem 1.

4. Framework

This section describes how to construct the Ψ function and the C array of T in $O(n \log\log |\mathcal{A}|)$ time.

We first assume that the length of T is a multiple of $2^{\lceil \log\log_{|\mathcal{A}|} n \rceil + 1}$. (Otherwise, we add enough \$ and a \$' at the end of T, where \$' is a character alphabetically smaller than the other characters in T, and proceed with the algorithm. The Ψ of this modified string is later converted into the Ψ of T in $O(n)$ time.) Let h be $\lceil \log\log_{|\mathcal{A}|} n \rceil$. For $0 \le k \le h$, we define T^k to be the string over the alphabet $\mathcal{A}^{2^k}$, which is formed by concatenating every 2^k characters in T to make one character. That is, $T^k[i] = T[(i-1)\cdot 2^k + 1..i \cdot 2^k]$, for $1 \le i \le n/2^k$. By definition, $T^0 = T$.

The basic framework is to use a bottom-up approach to construct Ψ_{T^i} for $i = \lceil \log\log_{|\mathcal{A}|} n \rceil$ down to 0, thereby obtaining Ψ. The details of each step are described as follows.

1. For $i = \lceil \log\log_{|\mathcal{A}|} n \rceil$
   (a) Construct suffix tree for T^i.
   (b) Construct Ψ_{T^i} from the suffix tree.
2. For $i = \lceil \log\log_{|\mathcal{A}|} n \rceil - 1$ to 0
   (a) Construct $\Psi_{T^i_e}$ based on $\Psi_{T^i_o}$. (Note: $\Psi_{T^i_o} = \Psi_{T^{i+1}}$.)
   (b) Construct Ψ_{T^i} based on $\Psi_{T^i_o}$ and $\Psi_{T^i_e}$.

Figure 3. The construction algorithm of Ψ_T.

For Step $i = \lceil \log\log_{|\mathcal{A}|} n \rceil$, Ψ_{T^i} is constructed by first building the suffix tree for T^i using Farach's algorithm [4], and then converting it back to Ψ_{T^i}.

For the remaining steps, we construct Ψ_{T^i} based on $\Psi_{T^{i+1}}$. Let m be the length of T^i (where m must be even), and Δ be the alphabet of T^i. For any string S, denote S_o and S_e to be the strings formed by merging every 2 characters in $S[1..m]$ and $S[2..m]S[1]$, respectively. More precisely, $S_o[i] = S[2i-1]S[2i]$ and $S_e[i] = S[2i]S[2i+1]$, where we set $S[m+1] = S[1]$. (Note that for $S = T^i$, the last characters of T^i_o and T^i_e are unique among the corresponding strings.) Intuitively, the suffixes of S_o and S_e correspond to the odd and the even suffixes of S, respectively. Figure 1(b) and (c) show an example for S_o and S_e. We have the following observation.

Observation 3 $\Psi_{T^i_o} = \Psi_{T^{i+1}}$.

This enables us to follow Farach's idea in the construction of suffix tree [4]. We first obtain $\Psi_{T^i_o} = \Psi_{T^{i+1}}$ from the previous step during the bottom-up algorithm. Then $\Psi_{T^i_e}$ is constructed based on T^i and $\Psi_{T^i_o}$. Finally, we merge $\Psi_{T^i_o}$ and $\Psi_{T^i_e}$ to give Ψ_{T^i}. The complete algorithm is shown in Figure 3.

Sections 5 and 6 describe in details how to obtain $\Psi_{T^i_e}$ from $\Psi_{T^i_o}$ and T^i in $O(m + |\Delta|)$ time, and how to merge $\Psi_{T^i_o}$ and $\Psi_{T^i_e}$ to obtain Ψ_{T^i} in $O(m \log\log|\Delta| + |\Delta|)$ time, respectively. Moreover, each of the above steps uses only $O(m \log|\Delta| + |\Delta|)$-bit space. This gives the following theorem.

Theorem 2 *The Ψ function and the C array of T can be constructed in $O(n \log\log|\mathcal{A}|)$ time and $O(n \log|\mathcal{A}|)$-bit space.*

Proof: We refer to the algorithm in Figure 3, which has two phases. For Phase 1, we have $i = \lceil \log\log_{|\mathcal{A}|} n \rceil$. We first construct the suffix tree for T^i whose size is $n/2^{\log\log_{|\mathcal{A}|} n} \le n \log|\mathcal{A}|/\log n$. This requires $O(n\log|\mathcal{A}|/\log n)$ time and $O(n \log|\mathcal{A}|)$-bit space by using Farach's suffix tree construction algorithm [4]. Then Ψ_{T^i} can be constructed in $O(n \log|\mathcal{A}|/\log n)$ time and $O(n\log|\mathcal{A}|)$-bit space. Thus, Phase 1 in total takes $O(n)$ time and $O(n \log|\mathcal{A}|)$-bit space.

For every Step i in Phase 2, we construct Ψ_{T^i}. Let Δ_i be the alphabet of T^i. Then, Part (a) takes $O(|T^i| + |\Delta_i|)$ time (Lemma 15), and Part (b) takes $O(|T^i|\log\log|\Delta_i| + |\Delta_i|)$ time (Lemma 17). For the space, both require $O(|T^i|\log|\Delta_i| + |\Delta_i|)$ bits. Note that $|T^i| = n/2^i$ and $|\Delta_i| \le |\mathcal{A}|^{2^i} \le n$, so the space used by Step i is $O(|T^i|\log|\Delta_i| + |\Delta_i|) = O(n\log|\mathcal{A}|)$ bits, and the time is $O(|T^i|\log\log|\Delta_i| + |\Delta_i|) = O(n/2^i(i + \log\log|\mathcal{A}|) + |\mathcal{A}|^{2^i})$. In total, the space for Phase 2 is $O(n \log|\mathcal{A}|)$ bits and the time is:

$$\begin{aligned} & \sum_{i=1}^{\lceil \log\log_{|\mathcal{A}|} n \rceil - 1} O\left(\frac{n}{2^i}(i + \log\log|\mathcal{A}|) + |\mathcal{A}|^{2^i}\right) \\ = \; & O(n \log\log|\mathcal{A}|). \end{aligned}$$

The whole algorithm for constructing Ψ therefore takes $O(n\log\log|\mathcal{A}|)$ time and $O(n\log|\mathcal{A}|)$-bit space. Finally, the C array can be constructed from Ψ using Lemma 7 in $O(n)$ time and $O(n\log|\mathcal{A}|)$-bit space. This completes the proof. □

Combining the above theorem with Lemmas 2, 3 and 4, we obtain the following:

Theorem 3 *Suffix array, compressed suffix array and FM-index can be computed in $O(n \log\log|\mathcal{A}|)$ time and $O(n\log|\mathcal{A}|)$-bit working space.*[7]

7 For FM-index, we assume that $|\mathcal{A}|\log|\mathcal{A}| = O(\log n)$. There is no restriction in the alphabet size for suffix array or compressed suffix array.

Theorem 4 *Suffix tree and compressed suffix tree can be computed in $O(n \log^{\epsilon} n)$ time and $O(n \log |\mathcal{A}|)$-bit working space, for any fixed ϵ with $0 < \epsilon < 1$.*

5. Constructing Ψ_{S_e}

Given $S[1..m]$ and Ψ_{S_o}, this section describes how to construct Ψ_{S_e}. Our approach is indirect, as prior to obtaining Ψ_{S_e}, we need to construct its compact representation C_e.

Let Δ be the alphabet of S. Define $x[1..m/2]$ to be an array of characters such that $x[i] = S[2SA_o[i] - 2]$ where $2SA_o[i] - 2$ is computed in modulo-n arithmetic. Let X_i be the string $x[i]S_o[SA_o[i]..m/2]S[1]$.

Observation 4 *X_i is a suffix of S_e if $SA_o[i] \neq 1$. Otherwise, $x[i] = S[m]$ which is unique among other characters in S.*

Let X be the set $\{X_k | 1 \leq k \leq m/2\}$. Intuitively, X is the same as the set of suffixes of S_e. See Figure 4(a) for an example of X_i.

		X_i		
i	$y[i]$	$x[i]$	$S_o[SA_o[i]..m/2]$	$S[1]$
1	bb	a	aa b$	a
2	ab	$	ab ba aa b$	a
3	aa	a	b$	a
4	$a	b	ba aa b$	a

(a)

i	$y \to C_e$	$x \to$ sorted x
1	ab	$
2	bb	a
3	aa	a
4	$a	b

(b)

Figure 4. Consider $S =$ abbaaab$. (a) The relationship between $x[i]$, $y[i]$ and X_i. (b) The array x and y after stable sorting on x.

Lemma 12 *The stable sorting order of $x[i]$ in x equals the rank of X_i in X.*

Proof: By omitting the first characters of every X_i's, they are of the form $S_o[SA_o[i]..m/2]S[1]$, which are already sorted. Thus, the rank of X_i is equal to the stable sorting order of $x[i]$ in x. □

Lemma 13 *Given Ψ_{S_o} and S, we can construct C_e in $O(m + |\Delta|)$ time and $O(m \log |\Delta| + |\Delta|)$-bit space.*

```
MergeCSA
```

1. Construct the augmented-Ψ_{S_o} and the augmented-Ψ_{S_e}.
2. Construct the auxiliary data structure for augmented-Ψ_{S_o} and that for augmented-Ψ_{S_e} to support efficient backward search.
3. Backward search for S_o on augmented-Ψ_{S_o} and augmented-Ψ_{S_e} simultaneously, and
   (a) at Step i, we obtain the rank of $S_o[m/2 - i + 1..m/2]$ among the odd suffixes and that among the even suffixes of S. Let the sum of the ranks be k.
   (b) Set $C[k] = S[m - 2i + 1]$.
4. Backward search for S_e on augmented-Ψ_{S_o} and augmented-Ψ_{S_e} simultaneously, and fill in $C[k]$ similarly.

Figure 5. Merging Ψ_{S_o} and Ψ_{S_e}.

Proof: *(sketch.)* Let $y[1..m]$ be an array such that $y[i]$ stores the two characters that immediately precedes $x[i]$ in S (i.e., $S[2SA_o[i] - 4]S[2SA_o[i] - 3]$). In fact, $y[i]$ is the preceding character of X_i in S_e. Using similar approach as in Lemma 7, x and y can be computed in $O(m)$ time, and both arrays occupy $O(m \log |\Delta|)$ bits.

To construct C_e, we perform a stable sort on x as in Lemma 6, and iteratively compute the stable sorting order k of $x[i]$, which is equal to the rank of X_i by Lemma 12. Therefore, we set $C_e[k] = y[i]$. The total time is $O(m+|\Delta|)$ and the total space is $O(m \log |\Delta| + |\Delta|)$ bits.[8] See Figure 4 for an example. □

Lemma 14 *Ψ_{S_e} can be constructed from C_e in $O(m+|\Delta|)$ time and $O(m \log |\Delta| + |\Delta|)$-bit space.*

Proof: The proof is similar to Lemma 6. □

Thus, we conclude this section with the following lemma.

Lemma 15 *Given Ψ_{S_o} and S, we can construct Ψ_{S_e} in $O(m + |\Delta|)$ time and $O(m \log |\Delta| + |\Delta|)$-bit space.*

Proof: The construction is as follows. First, C_e is constructed from Ψ_{S_o} and S by Lemma 13. Then, Ψ_{S_e} is constructed from C_e by Lemma 14. The lemma thus follows. □

8 A brute force approach requires $O(|\Delta| \log m)$-bit space for the stable sorting table. The space can be reduced to $O(m \log |\Delta|)$ bits under careful management, with the help of an $O(m + |\Delta|)$-bit vector supported by constant time *rank* and *select* queries.

6. Merging Ψ_{S_o} and Ψ_{S_e}

In this section, we construct Ψ_S from Ψ_{S_o} and Ψ_{S_e}. The idea is to determine the rank of any suffix of S among all suffixes of S, and based on this information, we construct the C array for Ψ_S. Finally, we convert C to Ψ_S by Lemma 6.

Let s be any suffix of S. Observe that the rank of s among the suffixes of S, is equivalent to the sum of the rank of s among the odd suffixes and that among the even suffixes of S. Based on this observation, we can construct the C array for Ψ_S as follows.

Firstly, we need to create the augmented-Ψ_{S_o} and the augmented-Ψ_{S_e}. Then, we construct the auxiliary data structure of Lemma 9 for the augmented-Ψ_{S_o} and that for the augmented-Ψ_{S_e}. Next, we perform backward searches for S_o on augmented-Ψ_{S_o} and augmented-Ψ_{S_e} simultaneously by Lemma 8. At each backward search step i, we get the rank (say, k) of $S_o[m/2 - i + 1..m/2]$ among the suffixes of S by computing the sum of its ranks among the odd suffixes and even suffixes, respectively. Then, $C[k]$ equals $S[m - 2i + 1]$, which is the preceding character of the suffix $S[m - 2i + 2..m] = S_o[m/2 - i + 1..m/2]$.

Similarly, we perform a simultaneous backward search for S_e on augmented-Ψ_{S_o} and augmented-Ψ_{S_e} to complete the remaining entries of C. Thus, we obtain C by $O(m)$ backward search steps. The algorithm is depicted as `MergeCSA` in Figure 5.

Lemma 16 *The algorithm* `MergeCSA` *correctly constructs* $C[1..m]$.

Proof: Recall that for every suffix $S[i..m]$, $C[SA^{-1}[i]]$ equals the preceding character of $S[i..m]$. For every odd suffix $S[i..m] = S_o[\lceil i/2 \rceil..m']$, Step 2 computes its rank k among all odd and even suffixes. By definition, $k = SA^{-1}[i]$. Therefore, Step 2 correctly assigns $C[k]$ to be the preceding character of $S[i..m]$. By the same argument, Step 3 correctly assigns $C[SA^{-1}[i]]$ to be the preceding character of $S[i..m]$.

Note that $\{SA^{-1}[i] \mid i = 1, \ldots, m\} = \{1, \ldots, m\}$. Therefore, after Steps 2 and 3, `MergeCSA` completely constructs $C[1..m]$. □

By Lemma 9, the auxiliary data structures can be constructed in $O(m + |\Delta|)$ time and $O(m + |\Delta|)$-bit space, and then each backward search step is done in $O(\log\log|\Delta|)$ time. On the other hand, the augmented-Ψ functions can be created in $O(m)$ time, and the total space required is $O(m \log |\Delta|)$ bits. Thus, we have the following lemma.

Lemma 17 *Given Ψ_{S_o} and Ψ_{S_e}, we can construct Ψ_S in $O(m \log\log|\Delta| + |\Delta|)$ time and $O(m \log|\Delta| + |\Delta|)$-bit space.*

7. Concluding Remarks

We have shown that suffix tree, suffix array, and other full-text indices can be constructed in $o(n \log n)$ time and $o(n \log n)$-bit space. This breaks the time-and-space barrier of a long-standing open problem.

Very recently linear-time algorithms for constructing suffix arrays have been proposed [14–16]. Though using interesting techniques, their algorithms require $O(n \log n)$-bit working space, and they will imply only $O(n \log^{\epsilon} n)$ time algorithms for constructing suffix arrays if the working space is limited to $O(n \log |\mathcal{A}|)$ bit. Therefore the algorithm in this paper will be essential for linear-time construction.

One of the open problem remains is that, whether we can construct suffix tree in optimal $O(n)$ time, while using optimal $O(n \log |\mathcal{A}|)$-bit working space. Another direction of research is to further reduce the working space for constructing full-text indices, from $O(n \log |\mathcal{A}|)$ bits to an input-dependent $O(nH)$ bits where H is the entropy of the input text.

Acknowledgment

The authors would like to thank Roberto Grossi, Tak-Wah Lam and Takeshi Tokuyama for helpful comments, and Ankur Gupta for sending us his paper. The work of the first author is supported in part by the Hong Kong RGC Grant HKU-7024/01E. The work of the second author is supported in part by the Grant-in-Aid of the Ministry of Education, Science, Sports and Culture of Japan. The work of the third author is supported in part by the NUS Academic Research Grant R-252-000-119-112.

References

[1] P. Beame and F. E. Fich. Optimal Bounds for the Predecessor Problem. In *Proceedings of Symposium on Theory of Computing*, pages 295–304, 1999.

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

[3] A. Crauser and P. Ferragina. A Theoretical and Experimental Study on the Construction of Suffix Arrays in External Memory. *Algorithmica*, 32:1–35, 2002.

[4] M. Farach. Optimal Suffix Tree Construction with Large Alphabets. In *Proceedings of Symposium on Foundations of Computer Science*, pages 137–143, 1997.

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

[6] 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.

[7] R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. Manuscript (Submitted for publication), 2001.

[8] D. A. Grossman and O. Frieder. *Information Retrieval: Algorithms and Heuristics*. Kluwer Academic Publishers, Boston, 1998.

[9] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, New York, 1997.

[10] T. Hagerup. Sorting and Searching on the Word RAM. In *Proceedings of Symposium on Theory Aspects of Computer Science*, pages 366–398, 1998.

[11] T. Hagerup, P. B. Miltersen, and R. Pagh. Deterministic Dictionaries. *Journal on Algorithms*, 41(1):69–85, 2001.

[12] W. K. Hon and K. Sadakane. Space-Economical Algorithms for Finding Maximal Unique Matches. In *Proceedings of Symposium on Combinatorial Pattern Matching*, pages 144–152, 2002.

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

[14] J. Kärkkäinen and P. Sanders. Simple Linear Work Suffix Array Construction. In *Proceedings of International Colloquium on Automata, Languages and Programming*, pages 943–955, 2003.

[15] D. Kim, J. Sim, H. Park, and K. Park. Linear-Time Construction of Suffix Arrays. In *Proceedings of Symposium on Combinatorial Pattern Matching*, pages 186–199, 2003.

[16] P. Ko and S. Aluru. Space Efficient Linear Time Construction of Suffix Arrays. In *Proceedings of Symposium on Combinatorial Pattern Matching*, pages 200–210, 2003.

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

[18] 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.

[19] U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

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

[21] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Springer-Verlag, Heidelberg, 1984.

[22] J. I. Munro. Tables. In *Proceedings of Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 37–42, 1996.

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

[24] K. Sadakane. Succinct representations of *lcp* information and improvements in the compressed suffix arrays. In *Proceedings of Symposium on Discrete Algorithms*, pages 225–232, 2002.

[25] K. Sadakane. New Text Indexing Functionalities of the Compressed Suffix Arrays. *Journal of Algorithms*, in press.

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

[27] 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.

[28] E. Ukkonen. On-line Construction of Suffix Trees. *Algorithmica*, 14(3):249–260, 1995.

[29] P. Weiner. Linear Pattern Matching Algorihms. In *Proceedings of Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[30] D. E. Willard. Log-Logarithmic Worst-Case Range Queries are Possible in Space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983.

[31] J. Zobel, A. Moffat, and K. Ramamohanarao. Guidelines for Presentation and Comparison of Indexing Techniques. *SIGMOD Record*, 25(3):10–15, 1996.