Improving Lookup Time Complexity of Compressed Suffix Arrays using Multi-ary Wavelet Tree

Zheng Wu
Department of Computer Science and Engineering, Pusan National University, Korea

Joong Chae Na
Department of Computer Science and Engineering, Sejong University, Korea

Minhwan Kim
Department of Computer Science and Engineering, Pusan National University, Korea

Dong Kyue Kim[†]
Department of Electronics and Communication Engineering, Hanyang University, Korea

Received 22 November 2008; Accepted 13 March 2009

In a given text T of size n, we need to search for the information that we are interested. In order to support fast searching, an index must be constructed by preprocessing the text. Suffix array is a kind of index data structure. The compressed suffix array (CSA) is one of the compressed indices based on the regularity of the suffix array, and can be compressed to the k^{th} order empirical entropy. In this paper we improve the lookup time complexity of the compressed suffix array by using the multi-ary wavelet tree at the cost of more space. In our implementation, the lookup time complexity of the compressed suffix array is $O(\log_\sigma^{\varepsilon/(1-\varepsilon)} n \log_r \sigma)$, and the space of the compressed suffix array is $\varepsilon^{-1} nH_k(T) + O(n \log\log n/\log_\sigma^\varepsilon n)$ bits, where σ is the size of alphabet, H_k is the kth order empirical entropy, r is the branching factor of the multi-ary wavelet tree such that $2 \le r \le \sqrt{n}$ and $r \le O(\log_\sigma^{1-\varepsilon} n)$, and $0 < \varepsilon < 1/2$ is a constant.

Categories and Subject Descriptors: E.1 [**Data Structures**]: Arrays; trees; E.4 [**Coding and Information Theory**]: Data compaction and compression; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems–*pattern matching; sorting and searching*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval–*search process*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Compressed suffix arrays, entropy, rank and select, text

---

†: corresponding author

compression, text indexing, wavelet trees

1. INTRODUCTION

1.1 Backgrounds

With the fast development of the Internet and computer technologies, the amount of the electronic data is growing at an exponential rate. However, as a result, retrieving useful data efficiently and exactly becomes a big challenge. Therefore, the *index* data structure is required to help finding the useful information that we need.

The most basic task of extracting information from text is *string matching*. String matching is the process of finding the occurrences of a short string that is called a *pattern* inside a text. Two classical *indices* for string matching are the *suffix tree* [McCreight 1976] and the *suffix array* [Manber and Myers 1993], which permit finding all the occurrences of any patterns without scanning the text sequentially. However, the space requirements are from 4 to 20 times the text size. Therefore, the space of the index obviously becomes the toughest problem.

A new trend in designing index focuses on compressing the index while permitting fast access to the index at the same time. As a result, designing a *compressed index* is actually to compress an index in order to reduce its space while at the cost of more time to access it. Therefore, a lot of work has been done to obtain various tradeoffs between the space taken by the index and the time to access the original index. We denote accessing the original index by the term *lookup*.

The leading research work on compressed indices is represented by the *compressed suffix array*, the *FM-index* [Ferragina and Manzini 2005; Ferragina et al. 2007] and the *compact suffix array* [Makinen 2003; Makinen and Navarro 2004], which support the functionalities of suffix arrays and suffix trees while stored in compressed form. All of the three indices take advantage of diffierent regularities of the suffix array to achieve compressibility. The compressed suffix array [Grossi and Vitter 2006] defines the *function* Ψ to represent the suffix array in compressed form while permitting fast access to the suffix array. The compressed suffix array was developed into a self-index and is related to the 0th empirical entropy [Sadakane 2002; Sadakane 2003].

The most recent result of compressed suffix arrays was given by Grossi et al. [Grossi et al. 2003]. In their main result, they show that the space of compressed suffix arrays can be related to the kth order empirical entropy by introducing a new partition scheme of function Ψ. In their implementation, they first do a recursive suffix array decomposition to obtain conceptual data structures of compressed suffix arrays. Then they partition the function Ψ at each maintained level of the recursive decomposition. Finally they implement these function Ψ at all of the maintained levels and some other data structures using *compressed representations of binary sequences* supporting *rank* and *select* [Pagh 1999; Raman et al. 2002]. Moreover, they introduce the *wavelet tree* to implement the function Ψ of the first recursive decomposition and maintain implementations of other levels unchanged in order to obtain a more compact space complexity of compressed suffix array at the cost of more lookup time. In this space/time tradeoff, given a text T of size n, they implement compressed suffix arrays using $\varepsilon^{-1}nH_k(T)+O(n \log\log n/\log_\sigma^\varepsilon n)$ bits of space and

Table I. Comparison of complexities between a previous result and our result on compressed suffix arrays. α and ε are constants such that $0 < \alpha < 1$ and $0 < \varepsilon < 1/2$.

CSA	Space (bits)	Lookup Time	Conditions
CSA [Grossi et al. 2003]	$\varepsilon^{-1}nH_k + O\left(\frac{n \log\log n}{\log_\sigma^\varepsilon n}\right)$	$O\left(\log_\sigma^{\frac{\varepsilon}{1-\varepsilon}} n \log \sigma\right)$	$k \le \alpha \log_\sigma n$
CSA (Proposed)	$\varepsilon^{-1}nH_k + O\left(\frac{n \log\log n}{\log_\sigma^\varepsilon n}\right)$ $+O\left(\frac{r \log\log n}{\log_\sigma n}\right)$	$O\left(\log_\sigma^{\frac{\varepsilon}{1-\varepsilon}} n \log_r \sigma\right)$	$k \le \alpha \log_\sigma n - 1$ $2 \le r \le \sqrt{n}$

permitting $O(\log_\sigma^{\varepsilon/(1-\varepsilon)} n \log\sigma)$ lookup time, where σ is the size of the alphabet, H_k is the k^{th} order empirical entropy, and ε is a constant such that $0 < \varepsilon < 1/2$. This tradeoff is shown in the first row of Table I.

1.2 Our Contributions

In this paper we improve the lookup time complexity of the implementation of compressed suffix arrays given by Grossi et al. [Grossi et al. 2003]. We use multi-ary wavelet trees [Ferragina et al. 2007] to implement function Ψ of the first level of the recursive decomposition in compressed suffix arrays that was previously implemented using binary wavelet trees, and maintain the implementation of other parts of the structure unchanged. Our main contribution is an implementation of compressed suffix arrays using $\varepsilon^{-1}nH_k(T)+O(n \log\log n/\log_\sigma^\varepsilon n)$ bits of space while supporting $O(\log_\sigma^{\varepsilon/(1-\varepsilon)} n \log_r \sigma)$ lookup time, where r is the branching factor of the multi-ary wavelet tree such that $2 \le r \le \sqrt{n}$ and $r \le O(\log_\sigma^{1-\varepsilon} n)$.

The general form of our result is shown in the second row of Table I. We improve the previous lookup time complexity $O(\log_\sigma^{\varepsilon/(1-\varepsilon)} n \log\sigma)$ to $O(\log_\sigma^{\varepsilon/(1-\varepsilon)} n \log_r \sigma)$ at the cost of $O(r \log\log n/\log_\sigma n)$ bits of more space, where $2 \le r \le \sqrt{n}$ and this condition is due to the multi-ary wavelet tree. When $2 \le r \le \sqrt{n}$ and $r \le O(\log_\sigma^{1-\varepsilon} n)$, $O(r \log\log n/\log_\sigma n)$ can be absorbed into the $O(n \log\log n/\log_\sigma^\varepsilon n)$, and thus our main result described above is obtained. When $2 \le r \le \sqrt{n}$ and $r > O(\log_\sigma^{1-\varepsilon} n)$, the compressed suffix array takes $\varepsilon^{-1}nH_k(T)+O(r \log\log n/\log_\sigma n)$ bits of space while supporting $O(\log_\sigma^{\varepsilon/(1-\varepsilon)} n \log_r \sigma)$ lookup time. In this result we improve the lookup time complexity further for increased value of r, while at the cost of larger space complexity.

Moreover, when the alphabet is $O(polylog(n))$ which is a reasonable size of alphabet for natural languages, the space of the compressed suffix array can be bounded by $\varepsilon^{-1}nH_k(T)+O(n \log\log n/\log_\sigma^\varepsilon n)+O(n/\log^\mu n)$ bits and the lookup time complexity is $O(\log_\sigma^{\varepsilon/(1-\varepsilon)} n)$, where μ is a constant such that $0 < \mu < 1$. The space can be bounded by $\varepsilon^{-1}nH_k(T)+O(n \log\log n/\log_\sigma^\varepsilon n)$ bits when $\varepsilon \le \mu < 1$.

2. PRELIMINARIES

2.1 The Empirical Entropy of a Text

Let T denote a text of size n. Each character of T belongs to an ordered alphabet $\Sigma = \{\alpha_1, \ldots, \alpha_\sigma\}$ of size σ. Let $T[i]$ be the i^{th} character in T, T_i be the suffix of T starting from position i, and $T_{i,j}$ be the substring of T starting at position i and ending at position j. $T[n]$ = # is a special character in Σ that only occurs once and lexicographically larger than any other characters in Σ. Let n_i denote the number of occurrences of the character α_i in T.

The 0^{th} *order empirical entropy* of T [Manzini 2001] is defined as

$$H_0(T) = -\sum_{i=1}^{\sigma} \frac{n_i}{n} \log \frac{n_i}{n},$$

where we assume 0 log 0 = 0 (all logarithms are taken to base 2 in this paper). The value of $nH_0(T)$ represents the maximum compression we can achieve by using a fixed codeword that can be uniquely decoded to each alphabet character.

We can achieve greater compression by choosing the codeword of α_i depending on the k characters preceding it (we call these k characters a context, where k is a constant). For any length-k context $x \in \Sigma^k$, let x_T be the concatenation of the single characters following each occurrence of x inside T. The k^{th} *order empirical entropy* of T [Manzini 2001] is defined as

$$H_k(T) = \frac{1}{n} \sum_{x \in \Sigma^k} |x_T| H_0(x_T).$$

The value of $nH_k(T)$ represents a lower bound to the compression we can achieve using codes which depend on their length-k contexts.

2.2 The Suffix Array and Function Ψ

The *suffix array* [Manber and Myers 1993] is an array $SA[1, n]$ which contains all starting positions of the suffixes of the text T such that $T_{SA[1]} \prec T_{SA[2]} \prec T_{SA[3]} \prec \cdots \prec T_{SA[n]}$, where "$\prec$" represents the lexicographical order between strings, i.e., the

i:	SA[i]	Ψ[i]	suffix $T_{SA[i]}$
1:	2	6	abbdaccbdbadca#
2:	6	12	accbdbadca#
3:	12	15	adca#
4:	15	16	a#
5:	11	3	badca#
6:	3	7	bbdaccbdbadca#
7:	4	13	bdaccbdbadca#
8:	9	14	bdbadca#
9:	1	1	cabbdaccbdbadca#
10:	14	4	ca#
11:	8	8	cbdbadca#
12:	7	11	ccbdbadca#
13:	5	2	daccbdbadca#
14:	10	5	dbadca#
15:	13	10	dca#
16:	16	16	#

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T	=	c	a	b	b	d	a	c	c	b	d	b	a	d	c	a	#

Figure 1. The suffix array and function Ψ of text T = *cabbdaccbdbadca#*.

suffix array gives the lexicographical order of all suffixes of the text T. The suffix array takes $O(n \log n)$ bits of space. Given a pattern P of size p, the suffix array answers the number of occurrences of P in $O(p \log n)$ time. An example of the suffix array is shown in Figure 1. The core concept of compressed suffix arrays is based on the regularity of the suffix array – the function Ψ [Grossi and Vitter 2006; Sadakane 2002; 2003; Grossi et al. 2003], which maps suffix $T_{SA[i]}$ to suffix $T_{SA[i]+1}$ in the suffix array and thus enables scanning the text through the indices of the suffix array in forward direction. Its formal definition is as follows:

$$\Psi(i) = j, \text{ such that } SA[j] = SA[i] \pmod{n}+1.$$

An example of function Ψ is shown in Figure 1.

We introduce regularities of the function Ψ that can be used in compressing Ψ. Since suffixes are sorted in SA, we can partition SA into at most σ intervals according to the first character of each suffix $T_{SA[i]}$. For character $y \in \Sigma$, we define *y-list* such that $\Psi(i)$ is in *y-list* if the first character of $T_{SA[i]}$ is y. For example, in Figure 2, observe that, for $5 \leq i \leq 8$, $\Psi(i)$ belong to b-list because the first character of suffixes $T_{SA[i]}$ is b. Each y-list can further be partitioned into sublists $\langle x, y \rangle$ using a k-character prefix x (length-k context x) of each suffix $T_{\Psi(i)}$ in y-list. An example of this partition scheme is shown in Figure 2, when $k = 1$. Note that yx is a prefix of $T_{SA[i]}$ for entries in $\langle x, y \rangle$ and all entries in the row corresponding to a context x form a contiguous index interval of the suffix array [Grossi et al. 2003].

2.3 Compressed Representations of Sequences

Most all compressed full-text indices take advantage of *compressed representations of sequences* which support *rank* and *select* operations on it. Given a general sequence T of length n, where $T[i] = c$ and $c \in \Sigma$, we compress it while supporting the following operations:

- $T[i]$: accesses the ith entry of T;
- $rank_c(T, i)$: returns the number of occurrences of character c in $T_{1,i}$;
- $select_c(T, j)$: returns the position of the jth occurrence of character c in T.

The simplified form is the compressed representation of binary sequences. One recent solution for representing compressed binary sequences is given by Pagh [Pagh 1999] and Raman et al. [Raman et al. 2002].

x	a list	b list	c list	d list	# list
a		3	1, 4	2	
b	6	7	8	5	
c	12		11	10	9
d	15	13, 14			
#	16				

Figure 2. An example of Partitioning Ψ according to its length-k contexts, when $k = 1$.

Lemma 2.1 [*Pagh 1999; Raman et al. 2002*] *Let* $B[1, n]$ *be a binary sequence containing t occurrences of bit 1. There exist a fully indexable dictionary (FID) that supports* $B[i]$, $rank_c(B, i)$ *and* $select_c(B, j)$ *in constant time using*

$$\log\binom{n}{t} + O\left(\frac{n \log\log n}{\log n}\right) = nH_0(B) + O\left(\frac{n \log\log n}{\log n}\right)$$

bits of space, where $c \in \{0, 1\}$.

Grossi et al. [Grossi et al. 2003] introduced the *wavelet tree* which is a balanced binary search tree. A lemma regarding the wavelet tree is given as follows:

Lemma 2.2 *[Grossi et al. 2003] Let* $T[1, n]$ *be a string over an arbitrary alphabet* Σ, *where* $|\Sigma| = \sigma$. *The wavelet tree built on T takes*

$$nH_0(T) + O\left(\frac{n \log\log n}{\log_\sigma n}\right)$$

bits of space and for any character $c \in \Sigma$ *and* $1 \le i \le n$, $1 \le j \le n$, *supports*

$$O(\log \sigma)$$

time $T[i]$, $rank_c(T, i)$ *and* $select_c(T, j)$ *operations.*

The wavelet tree has been extended to its generalized form – the multi-ary wavelet tree [Ferragina et al. 2007], as shown in Figure 3. The height of this r-ary tree is at most $1+\log_r \sigma$, which is smaller than that of the binary wavelet tree. Then the *access*, *rank* and *select* can be calculated in a similar way as in the binary wavelet tree.

Lemma 2.3 *[Ferragina et al. 2007] Let* $T[1, n]$ *be a sequence over an arbitrary alphabet* Σ, *where* $|\Sigma| = \sigma$. *The multi-ary wavelet tree built on T for* $2 \le r \le \min(\sigma, \sqrt{n})$ *takes*

$$nH_0(T) + O(\sigma \log n) + O\left(\frac{rn \log\log n}{\log_\sigma n}\right)$$

bits of space, and supports queries $T[i]$, $rank_c(T, i)$ and $select_c(T, j)$ *in*

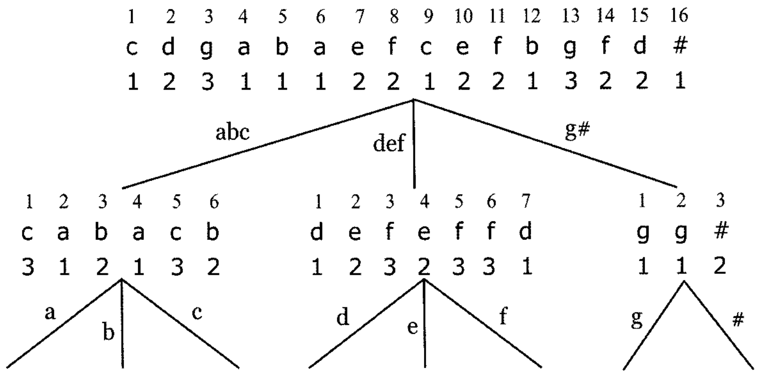


Figure 3. An example of the multi-ary wavelet tree, when $\sigma = 8$, $r = 3$.

$$O(\log_r \sigma)$$

time, for any $c \in \Sigma$ and $1 \le i, j \le n$.

If $\sigma = O(polylog(n))$, *then r can be chosen so that the resulting multi-ary wavelet tree takes*

$$nH_0(T) + O\left(\frac{n}{\log^{\mu} n}\right)$$

bits of space and supports all the three kinds of queries in constant time, for any constant $0 < \mu < 1$.

Note that it should be guaranteed that $r = o(\log n / \log \log n)$ [Ferragina et al. 2007].

3. COMPRESSED SUFFIX ARRAY

In this section we introduce the compressed suffix array (CSA) and a recent implementation [Grossi et al. 2003]. We focus on the data structures of CSA, the decomposition scheme, the space and lookup time complexity analysis. As for how to do different kinds of queries and how to achieve self-indexing, the descriptions are given in [Sadakane 2002; 2003] in detail, and are not covered here.

CSA contains the same information as the suffix array, which requires less space while at the cost of non-constant lookup time. A general introduction for CSA is as follows: given a text T of length n and its suffix array SA, the *compressed suffix array* for T supports the following operations without requiring explicit storage of T or SA:

– *compress*: produces a compressed representation that encodes T and SA;
– *lookup*: given $1 \le i \le n$, returns $SA[i]$;
– *substring*: decompresses the substring of T consisting of the first s characters (a prefix) of the suffix $T_{SA[i]}$, for $1 \le i \le n$ and $1 \le s \le n$. $SA_{[i]} + 1$.

We first introduce the basic structure of CSA and then one recent implementation.

3.1 Basic Structure of Compressed Suffix Arrays

Now we introduce the recursive decomposition scheme of CSA during which its structure is built. In the base case, we denote SA by SA_0, and let $n_0 = n$. For simplicity we assume that n is a power of 2. In the inductive phase $0 \le l \le h$ where h is the final phase, we start with suffix array SA_l, which is available by induction. SA_l stores a permutation of 1, 2,..., n_l, where $n_l = n/2^l$. The permutation results from sorting the suffixes of T whose starting positions are multiples of 2^l. That is, in each inductive phase $0 \le l < h$, given SA_l, we transform it into an equivalent but more succinct representation consisting of a binary sequence B_l, the function Ψ_l and a suffix array SA_{l+1}. We run three main steps as follows:

(1) Construct a binary sequence B_l of n_l bits supporting *rank*, such that $B[i] = 1$ if $SA_l[i]$ is even and $B[i] = 0$ if $SA_l[i]$ is odd;
(2) Construct function Ψ on SA_l, which is denoted by Ψ_l;
(3) Group together even text positions in SA_l and divide each of them by 2. The resulting values form a permutation of $\{1, 2, ..., n_{l+1}\}$, where $n_{l+1} = n_l/2 = n/2^{l+1}$. Store them in a new suffix array SA_{l+1} of length n_{l+1} and remove the old suffix array SA_l.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
SA_0 =	2	6	12	15	11	3	4	9	1	14	8	7	5	10	13	16
B_0 =	1	1	1	0	0	0	1	0	0	1	1	0	0	1	0	1
Ψ_0 =	6	12	15	16	3	7	13	14	1	4	8	11	2	5	10	9

	1	2	3	4	5	6	7	8
SA_1 =	1	3	6	2	7	4	5	8
B_1 =	0	0	1	1	0	1	0	1
Ψ_1 =	4	6	5	2	8	7	3	1

	1	2	3	4
SA_2 =	3	1	2	4
B_2 =	0	0	1	1
Ψ_2 =	4	3	1	2

Figure 4. An example of suffix array decomposition.

Figure 4 illustrates the process of the recursive decomposition scheme. We store B_l and Ψ_l for $0 \le l < h$, and SA_h for $l = h$. Note that SA_l is not stored for $0 \le l < h$.

Given B_l and Ψ_l for $0 \le l < h$, and SA_h stored for $l = h$, the original $SA[i]$ can be retrieved in $O(h)$ time for any $1 \le i \le n$ if both B_l and Ψ_l can be accessed in constant time. Lookup for $SA_l[i]$ can be answered as follows:

$$SA_l[i] = \begin{cases} 2 \cdot SA_{l+1}[rank_1(B_l, i)] & \text{if } B_l[i]=1 \\ SA_l[\Psi_l(i)]-1 & \text{if } B_l[i]=0 \end{cases}$$

We introduce a basic implementation of CSA. We can keep a total of three levels: level 0, level $h' = \frac{1}{2}\log\log n$ and level $h = \log\log n$. The problem is how to reconstruct SA_0 from $SA_{h'}$ and how to reconstruct $SA_{h'}$ from SA_h. Since there are $n_{h'}$ entries of SA_0 stored into $SA_{h'}$, we redefine B_0 of length n such that its 1 bits mark the positions of SA_0 that are maintained in $SA_{h'}$. Similarly we redefine $B_{h'}$.

In order to retrieve $SA_0[i]$, we use Ψ_0 to walk along indices i', i'', ..., such that $SA_0[i]+1 = SA_0[i']$, $SA_0[i']+1 = SA_0[i'']$, and so on, until we reach an index marked by B_0. Let s be the number of steps in the walk and r be the rank of the index thus found in B_0. We switch to level h' and reconstruct the r^{th} entry at level h' from the explicit representation of SA_h by a similar walk until we find an index marked by $B_{h'}$. Let s' be the number of steps in the latter walk and r' be the rank of the index thus found in $B_{h'}$. Then $SA_0[i] = SA_h[r'] \cdot 2^h + s' \cdot 2^{l'} + s \cdot 2^0$, which means the lookup time complexity is $O(\sqrt{\log n})$.

3.2 Implementation using Wavelet Trees

Now we describe an implementation given by Grossi et al. [Grossi et al. 2003]. In order to store the final data structures in a more general setting, we maintain the following levels:

– Level 0 is the same as before;

– The last level $h = \log\log n - \varepsilon \log\log_\sigma n - \log\log\log n$, where $0 < \varepsilon < 1/2$ is a constant;

– Level $h' = \log\log_\sigma n$ between level 0 and level h;

– One level every other $\gamma h'$ levels between level 0 and level h', where $\gamma = \varepsilon/(1-\varepsilon)$ with

Table II. The space complexities of all data structures of compressed suffix arrays.

Information	Data Structure	Space Complexity (bits)
Ψ_0	Wavelet Tree	$nH_k + O\left(\frac{n \log \log n}{\log_\sigma n}\right)$
Ψ_l for $0 \le l \le h'$	FID	$\frac{1}{\gamma} nH_k + O\left(\frac{n \log \log n}{\log^\varepsilon n}\right)$
B_l for $0 < l \le h'$	FID	$O\left(\frac{n \log \log n}{\log_\sigma^\varepsilon n}\right)$
SA_h	Integer Array	$O\left(\frac{n \log \log n}{\log_\sigma^\varepsilon n}\right)$

$0 < \gamma < 1$, i.e., $\gamma^{-1} + 1 = \varepsilon^{-1}$.

That is, we maintain $\varepsilon^{-1} - 1$ levels between level 0 and level h' (including boundary levels), and ε^{-1} levels in total.

Thus we have to store the following information:

- Ψ_l and B_l for all $l = i\gamma h'$ where $0 \le i \le \frac{1}{\gamma}$;
- SA_h for $l = h$.

We show how to store these information efficiently. According to the regularity of the function Ψ, at each level l for $0 \le l \le h'$, Ψ_l forms σ^k contiguous intervals as described in Section 2.3. At level 0, each contiguous interval belonging to the same context x is implemented using a wavelet tree. At level $0 < l \le h'$, each sublist $\langle x, y \rangle$ is implemented using a FID. And each B_l for $0 \le l \le h'$ is also implemented using a FID respectively. Finally, at level h, SA_h is stored explicitly. A summary is given in Table II. By adding up the space of all data structures above, we get the following lemma:

Lemma 3.1 *[Grossi et al. 2003] The compressed suffix array can be implemented using*

$$\frac{1}{\varepsilon} nH_k(T) + O\left(\frac{n \log \log n}{\log_\sigma^\varepsilon n}\right)$$

bits of space, where $k \le \alpha \log_\sigma n$, $0 < \alpha < 1$, *so that lookup operation (accessing to the original suffix array entries) takes*

$$O\left(\log_\sigma^{\frac{\varepsilon}{1-\varepsilon}} n \log \sigma\right)$$

time, for any fixed value $0 < \varepsilon < \frac{1}{2}$.

4. COMPRESSED SUFFIX ARRAYS USING MULTI-ARY WAVELET TREES

In this section we give our proposition of CSA by using multi-ary wavelet trees to improve the lookup time complexity of CSA. We show that multi-ary wavelet trees can be applied to implement Ψ_0 so that lookup time complexity can be improved. We use the same suffix array decomposition scheme that is used in Section 3.2.

In order to get a general result, we consider the multi-ary wavelet tree without giving any assumptions on the alphabet Σ first. By utilizing multi-ary wavelet trees to implement Ψ_0 belonging to the same contexts, we can obtain that the following.

Theorem 4.1 *By utilizing multi-ary wavelet trees to implement Ψ_0, the compressed suffix array can be implemented using*

$$\frac{1}{\varepsilon}nH_k(T)+O\left(\frac{n\log\log n}{\log_\sigma^\varepsilon n}\right)$$

bits of space, when $r\le O(\log_\sigma^{1-\varepsilon}n)$, or using

$$\frac{1}{\varepsilon}nH_k(T)+O\left(\frac{r\log\log n}{\log_\sigma n}\right)$$

bits of space, when $r> O(\log_\sigma^{1-\varepsilon}n)$, where $k\le\alpha\log_\sigma n-1$, $0<\alpha<1$, so that accessing to the original suffix array entries takes

$$O\left(\log_\sigma^{\frac{\varepsilon}{1-\varepsilon}}n\ \log_r\sigma\right)$$

time, for any fixed value $0<\varepsilon<\frac{1}{2}$, and r is the branching factor of the multi-ary wavelet tree such that $2\le r\le\min(\sigma,\sqrt{n})$.

Proof. Our proposition differs from the original one only in the implementation of Ψ_0. We begin with proving the space of multi-ary wavelet trees for Ψ_0. For each context x, namely, a row in Figure 2, we use one multi-ary wavelet tree to encode Ψ_0 belonging to the same x. By Lemma 2.3 we have that the space taken by each row is

$$n^xH_0(x_{\Psi_0^x})+O(\sigma\log n^x)+O\left(\frac{rn^x\log\log n^x}{\log_\sigma n^x}\right)$$

bits of space, where n^x is the number of Ψ_0 entries in context x. Since there are at most σ^k contexts, by adding up the space we obtain that the space consumes at most

$$nH_k+O(\sigma^{k+1}\log n)+O\left(\frac{rn\log\log n}{\log_\sigma n}\right)$$

bits.

The first term is due to the definition of the kth order empirical entropy, that is

$$\sum_{x\in\Sigma^k}n^xH_0(x_{\Psi_0^x})=nH_k.$$

The second term holds as follows

$$\sum_{x\in\Sigma^k}O(\sigma\log n^x)\le O\left(\sigma\log\left(\frac{n_1+n_2+\ldots+n_{\sigma^k}}{\sigma^k}\right)^{\sigma^k}\right)=O(\sigma^{k+1}\log n).$$

To compare $O(\sigma^{k+1}\log n)$ and $O(n\log\log n/\log_\sigma^\varepsilon n)$, we have

$$\frac{O\left(\frac{n\log\log n}{\log_\sigma^\varepsilon n}\right)}{O(\sigma^{k+1}\log n)}\ge O\left(\frac{n^{1-\alpha}\log\log n\log^\varepsilon\sigma}{\log^{1+\varepsilon}n}\right)\ge O(\log\log n\log^\varepsilon\sigma).$$

The first inequality is due to the fact that $k<\alpha\log_\sigma n-1$ for $0<\alpha<1$. Thus $O(\sigma^{k+1}$

log n) can be absorbed into $O(n \log\log n/\log_\sigma^\varepsilon n)$.

The third term is due to that

$$\sum_{x \in \Sigma^k} O\left(\frac{rn^x \log\log n^x}{\log_\sigma n^x}\right) \le \sigma^k O\left(\frac{r\frac{n}{\sigma^k}\log\log\frac{n}{\sigma^k}}{\log_\sigma \frac{n}{\sigma^k}}\right) = O\left(\frac{rn\log\log n}{\log_\sigma n}\right).$$

Note that the inequality holds because $O(rn^x \log\log n^x/\log_\sigma n^x)$ is a concave function. To compare $O(rn \log\log n/\log_\sigma n)$ and $O(n \log\log n/\log_\sigma^\varepsilon n)$, we have that

$$\frac{O\left(\frac{rn\log\log n}{\log_\sigma n}\right)}{O\left(\frac{n\log\log n}{\log_\sigma^\varepsilon n}\right)} = O\left(\frac{r}{\log_\sigma^{1-\varepsilon} n}\right).$$

Therefore, when r is chosen to be $r \le O(\log_\sigma^{1-\varepsilon} n)$, $O(rn \log\log n/\log_\sigma n)$ can be absorbed into $O(n \log\log n/\log_\sigma^\varepsilon n)$; when $r > O(\log_\sigma^{1-\varepsilon} n)$, $O(n \log\log n/\log_\sigma^\varepsilon n)$ can be absorbed into $O(rn \log\log n/\log_\sigma n)$.

Because other data structures are the same as before, by adding the space of all of them together the space of Theorem 4.1 is proved.

The only remaining task is to compare the time of stepping from level 0 to level h' and that of stepping from level h' to h. The former one is

$$O\left(\log_\sigma^{\frac{\varepsilon}{1-\varepsilon}} n \log_r \sigma\right).$$

And the latter one is

$$2^{h-h'} = \frac{\log n}{\log_\sigma^{1-\varepsilon} n \log\log n} = \frac{\log_\sigma^\varepsilon n \log\sigma}{\log\log n} = \log_\sigma^\varepsilon n \log_{\log n}\sigma \le O\left(\log_\sigma^{\frac{\varepsilon}{1-\varepsilon}} n \log_r \sigma\right).$$

Therefore, the final lookup time complexity is $O(\log_\sigma^{\varepsilon/(1-\varepsilon)} n \log_r \sigma)$.
Note that this lookup time has been improved over Lemma 3.1 at the cost of more space □.

By bounding the alphabet to be $O(polylog(n))$, which is a reasonable alphabet size, we can prove the improvement to be as follows:

Corollary 4.1 If $\sigma = O(polylog(n))$, *then r can be chosen so that by utilizing the multi-ary wavelet tree to encode* Ψ_0 *values, the compressed suffix array can be implemented using*

$$\frac{1}{\varepsilon} nH_k + O\left(\frac{n\log\log n}{\log_\sigma^\varepsilon n}\right)$$

bits of space, when $\varepsilon \le \mu < 1$, *or using*

$$\frac{1}{\varepsilon} nH_k + O\left(\frac{n}{\log^\mu n}\right)$$

bits of space, when $0 < \mu < \varepsilon$, *where* $k \le \alpha \log_\sigma n$, $0 < \alpha < 1$, *so that lookup time* *(accessing to the original suffix array entries) takes*

$$O\left(\log_\sigma^{\frac{\varepsilon}{1-\varepsilon}} n\right)$$

time, for any fixed value $0 < \varepsilon < \frac{1}{2}$.

Proof. The analysis is similar to the proof of Theorem 4.1. That is, for each context x, each multi-ary wavelet tree requires

$$n^x H_0(x_{\Upsilon_0^x}) + O\left(\frac{n^x}{\log^\mu n^x}\right)$$

bits.

When adding up the space taken by multi-ary wavelet trees corresponding to all x,

$$\sum_{x \in \Sigma^k} \left(n^x H_0(x_T) + O\left(\frac{n^x}{\log^\mu n^x}\right)\right) \le nH_k + O\left(\frac{n}{\log^\mu n}\right)$$

bits of space is required.

Note that the inequality holds because $O(n^x/\log^\mu n^x)$ is a concave function.

To compare $O(n \log\log n/\log_\sigma^\varepsilon n)$ and $O(n/\log^\mu n)$, we obtain that

$$\frac{O\left(\frac{n \log\log n}{\log_\sigma^\varepsilon n}\right)}{O\left(\frac{n}{\log^\mu n}\right)} = O(\log\log n \log^\varepsilon \sigma \log^{\mu-\varepsilon} n) = O(\log\log^{1+\varepsilon} n \log^{\mu-\varepsilon} n).$$

Therefore, if $0 < \mu < \varepsilon$, $O(n \log\log n/\log_\sigma^\varepsilon n)$ can be absorbed into $O(n/\log^\mu n)$; in the other case if $\varepsilon \le \mu < 1$, $O(n/\log^\mu n)$ can be absorbed into $O(n \log\log n/\log_\sigma^\varepsilon n)$. By putting up all data structures together, the space in Corollary 4.2 is proved.

The last piece of proof is the time complexity when stepping from level h' to h, which can be obtained as the following:

$$2^{h-h'} = \frac{\log_\sigma^\varepsilon n \log\sigma}{\log\log n} = \frac{\log_\sigma^{\frac{\varepsilon}{1-\varepsilon}} n}{\log\log n}(\log\log n) = O\left((\log_\sigma n)^{\frac{\varepsilon}{1-\varepsilon}}\right) \ \square.$$

5. CONCLUSION

In this paper we focused on improving lookup time complexity of the compressed suffix array. We improve the lookup time complexity of compressed suffix arrays using multi-ary wavelet tree at the cost of more space. We proved that the compressed suffix array can be stored in $\varepsilon^{-1} nH_k(T) + O(n \log\log n/\log_\sigma^\varepsilon n)$ bits of space, where r is the branching factor of the multi-ary wavelet tree, so that lookup time of the compressed suffix array can be improved to $O(\log_\sigma^{\varepsilon/(1-\varepsilon)} n \log_r \sigma)$. When $\sigma = O(polylog(n))$, we showed that the space of the compressed suffix array is $\varepsilon^{-1} nH_k(T) + O(n \log\log n/\log_\sigma^\varepsilon n) + O(n/\log^\mu n)$ bits, where μ is a constant such that $0 < \mu < 1$, and the lookup time was improved to $O(\log_\sigma^{\varepsilon/(1-\varepsilon)} n)$.

ACKNOWLEDGMENTS

This research was supported by the MKE (The Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the IITA (Institute for Information Technology Advancement) (IITA-2009-(C1090-0902-0003)).

REFERENCES

FERRAGINA, P. and G. MANZINI. 2005. Index compressed texts. *J. Assoc. Comput. Mach.* 52(4): 552–581.

FERRAGINA, P., G. MANZINI, V. MAKINEN, and G. NAVARRO. 2007. Compressed representation of sequences and full-text indexes. *ACM Transactions on Algorithms* (TALG), 3(2):1–25.

GROSSI, R., A. GUPTA, and J. VITTER. 2003. High-order entropy-compressed text indexes. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA), 841–850.

GROSSI, R. and J. VITTER. 2006. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.* 35(2):378–407.

MAKINEN, V. 2003. Compact suffix array – a space-efficient full-text index. *Fund. Inform.* 56(1-2):191–210.

MAKINEN, V. and G. NAVARRO. 2004. Compressed compact suffix arrays. in *Proceedings of the 15th Annual Symposium on Combinational Pattern Matching* (CPM). Lecture Notes in Computer Science, vol. 3109. Springer-Verlag, Berlin, Germany, 420–433.

MANBER, U. and G. MYERS. 1993. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.* 22(5):935–948.

MANZINI, G. 2001. An analysis of the burrows-wheeler transform. *J. Assoc. Comput. Marc.* 48(3):407–430.

MCCREIGHT, E. 1976. A space-economical suffix tree construction algorithm. *J. Assoc. Comput. Marc.* 23(2):262–272.

PAGH, R. 1999. Low redundancy in dictionaries with o(1) worst case lookup time. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming* (ICALP), 595–604.

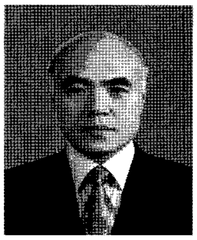
RAMAN, R., V. RAMAN, and S. RAO. 2002. Succinct indexable dictionaries with applications toencoding k-ary trees and multisets. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA), 233–242.

SADAKANE, K. 2002. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA), 225–232.

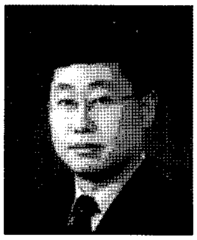
SADAKANE, K. 2003. New text indexing functionalities of the compressed suffix arrays. *J. Alg.* 48(2):294–313.

Zheng Wu He received an M.S. in the Dept. of Computer Science and Engineering in Pusan National University, Korea. He currently works as an engineer in LG Electronics, Korea.

Joong Chae Na He received a B.S., an M.S., and a Ph.D. in Computer Science and Engineering from Seoul National University in 1998, 2000, and 2005, respectively. He worked as a visiting postdoctoral researcher in the Department of Computer Science at the University of Helsinki in 2006. He is currently a professor in Department of Computer Science and Engineering at Sejong University. His research interests include design and analysis of algorithms, and bioinformatics.



Minhwan Kim He received his B.S., M.S., and Ph.D. degrees from Seoul National University, Seoul, Korea, in 1980, 1983, and 1988, respectively. He is currently a professor of the Dept. of Computer Science and Engineering in Pusan National University, Korea. His research interests include multimedia information retrieval, intelligent surveillance system, and computer vision.



Dong Kyue Kim He received his B.S., M.S., and Ph.D. degrees from Seoul National University, Seoul, Korea, in 1992, 1994, and 1988, respectively. He is currently an associate professor of the Dept. of Electronics and Communication Engineering in Hanyang University, Korea. His research interests are in the area of embedded security systems, crypto-coprocessors,, and theory of computation.