써픽스 배열 합병을 이용한 일반화된 써픽스 배열의 효율적인 구축 알고리즘
(Efficient Construction of Generalized Suffix Arrays by Merging Suffix Arrays)

전 정 은 [†]　　박 희 진 [††]　　김 동 규 [†††]
(Jeong Eun Jeon)　(Heejin Park)　(Dong Kyue Kim)

요 약 본 논문에서는 A와 B의 써픽스 배열이 주어졌을 때 두 배열을 합병하여 이들의 일반화된 써픽스 배열을 구축하는 방법을 연구하였다. 홀수 써픽스와 짝수 써픽스같이 특별한 경우의 두 써픽스를 합병하는 알고리즘은 이미 발표되었지만, A와 B가 임의의 문자열인 일반적인 경우 두 써픽스 배열을 합병하는 효율적인 알고리즘은 아직 개발되지 않았다. 따라서 현재까지는 A와 B의 써픽스 배열을 합병하기 위해서 A와 B의 써픽스 배열이 이미 주어져 있음에도 불구하고 $A\#B\$$라는 문자열에 대한 써픽스 배열을 다시 구축해야했다. 본 논문에서는 상수 문자집합이나 정수 문자집합에서 정의된 임의의 두 문자열 A와 B에 대한 써픽스 배열을 합병하는 효율적인 알고리즘을 제시한다. 실험결과 상수문자집합의 경우 $A\#B\$$에 대한 써픽스 배열을 다시 구축하는 것보다 합병하는 것이 5배 정도 빨랐다.

여기서 제시한 알고리즘은 써픽스 배열 A에서 스트링 B의 모든 써픽스를 검색하여야 한다. 이를 위해 써픽스 배열에서 정의한 써픽스 링크를 사용하였고, 또 써픽스 링크를 계산하는 효율적인 알고리즘도 개발하였다. 써픽스 링크는 생물정보학에서 사용되는 매칭 통계나 최장 공통 부분 문자열 검색처럼 다른 스트링의 써픽스 배열에서 주어진 스트링의 모든 써픽스를 찾는 데 이용할 수 있으므로, 이를 계산하는 효율적인 방법을 제시한 것 역시 많은 의미를 가진다. 실험을 통해 여기서 제시한 방법이 기존 알고리즘 중 가장 빠른 방법보다 3~4배 정도 빠르다는 것을 보였다.

키워드 : 써픽스 배열, 써픽스 배열 합병, 일반화된 써픽스 배열, 써픽스 링크, 매칭 통계 계산, 최장 공통 부분 문자열 검색

Abstract We consider constructing the generalized suffix array of strings A and B when the suffix arrays of A and B are given, i.e., merging two suffix arrays of A and B. There are efficient algorithms to merge some special suffix arrays such as the odd array and the even array. However, for the general case that A and B are arbitrary strings, no efficient merging algorithms have been developed. Thus, one had to construct the generalized suffix array of A and B by constructing the suffix array of $A\#S\$$ from scratch, even though the suffix arrays of A and B are given. In this paper, we present efficient merging algorithms for the suffix arrays of two arbitrary strings A and B drawn from constant and integer alphabets. The experimental results show that merging two suffix arrays of A and B are about 5 times faster than constructing the suffix array of $A\#B\$$ from scratch for constant alphabets.

Our algorithms include searching all suffixes of string B in the suffix array of A. To do this, we use suffix links in suffix arrays and we developed efficient algorithms for computing the suffix links. Efficient computation of suffix links is another contribution of this paper because it can be used to solve other problems occurred in bioinformatics that should search all suffixes of a given string in the suffix array of another string such as computing matching statistics, finding longest common substrings, and so on. The experimental results show that our methods for computing suffix links is about 3-4 times faster than the previous fastest methods.

· 이 논문은 2003년도 한국학술진흥재단의 지원에 의하여 연구되었음 (KRF-2003-003-D00343).
† 학생회원 : 부산대학교 컴퓨터공학과
jejun@islab.ce.pusan.ac.kr
†† 종신회원 : 한양대학교 정보통신학부 교수
hjpark@hanyang.ac.kr
††† 종신회원 : 부산대학교 컴퓨터공학과 교수
dkkim1@pusan.ac.kr
논문접수 : 2005년 1월 24일
심사완료 : 2005년 3월 24일

Key words : suffix arrays, merging suffix arrays, generalized suffix array, suffix link, computing matching statistics, finding longest common substrings

1. Introduction

The full-text index data structure for a text string incorporates the indices for all the suffixes of the text. The full-text index data structure is used in numerous applications[1]. For example, searching DNA sequences in a whole genome, one of the primary operations in bioinformatics, requires the full-text index data structure of the whole genome. When we consider the complexity of full-text index data structures, there are three types of alphabets from which text T of length n is drawn: (i) a constant alphabet, (ii) an integer alphabet where symbols are integers in the range $[0, n^c]$ for a constant c, and (iii) a general alphabet in which the only operations on string T are symbol comparisons.

Two fundamental full-text index data structures are the suffix tree and the suffix array[1]. The suffix tree is introduced earlier than the suffix array. The suffix tree due to McCreight[2] is a compacted trie of all suffixes of the string. It was designed as a simplified version of Weiner's position tree[3]. The suffix tree is time-efficient in that the suffix tree for a string of length n can be constructed in $O(n)$ time[2,4-6] and a pattern of length m can be searched in $O(m)$ time in the suffix tree if the size of alphabet is constant. However, it is not space-efficient because it consumes quite large space.

The suffix array is developed as a space-efficient alternative to the suffix tree. The suffix array due to Manber and Myers[7] and independently due to Gonnet et al.[8] is basically a sorted list of all the suffixes of the string. Since it is developed as a space-efficient full-text index data structure, it was not so time-efficient as the suffix tree when it was introduced. It took $O(n\log n)$ time for constructing the suffix array[1] and $O(m+\log n)$ time for pattern search even with the lcp (longest common prefix) information. However, researchers have tried to make the suffix array as time-efficient as the suffix tree. Recently, almost at the same time, several algorithms have been developed to directly construct the suffix array in $O(n)$ time[9-11]. In addition, searching a pattern in $O(m)$ time in suffix arrays have been achieved[12,13].

Let A and B denote strings of lengths n_a and n_b respectively. The generalized suffix tree (resp. array) of two strings A and B, is the suffix tree (resp. array) of the concatenated string $A\#B\$$. Generalized suffix trees and arrays are useful to solve various string processing problems occurring in bioinformatics[1] such as finding longest common substrings, recognizing DNA contamination, computing matching statistics, and so on. The simplest way to construct the generalized suffix tree (resp. array) is to construct the suffix tree (resp. array) for $A\#B\$$ from scratch. We can construct the suffix tree in $O(n_a+n_b)$ time for constant alphabet due to McCreight[2], Chen and Seiferas[14], and Ukkonen[6], and for integer alphabet due to Farach et al.[4,5]. Also, we can construct the suffix array in $O(n_a+n_b)$ time for constant and integer alphabets due to Kim et al.[10], Ko and Aluru[11], and Kärkkäinen and Sanders[9].

If the suffix trees of A and B are already constructed, we can construct the generalized suffix tree in a more efficient way: We construct the generalized suffix tree by inserting the suffixes of B, from the longest to the shortest, into the suffix tree of A (due to McCreight[2]), by inserting the suffixes of A, from the shortest to the longest, into the suffix tree of B (due to Chen and Seiferas[14]), or by merging the suffix trees of A and B using the structure of the suffix trees (due to Farach et al.[4,5]).

When it comes to merging two suffix arrays, efficient algorithms have been developed only for some special suffix arrays. Kim et al.[10] developed

1) The suffix array could be constructed in $O(n)$ time if we first constructed the suffix tree and then the suffix array from the suffix tree. However, constructing the suffix array in this way is not space-efficient.

merging algorithms for the odd and even arrays where the odd array is the sorted array of the odd suffixes of a string and the even array is that of the even suffixes of the string. Hon et al.[15] developed merging algorithms for the succinctly represented odd and even arrays. Kärkkäinen and Sanders[9] developed merging algorithms for two arrays such that one array is the sorted array of the suffixes of a string beginning at positions i mod $3=0$ and the other is that of the other suffixes of the string. However, to the best of our knowledge, it seems that no efficient merging algorithms have been developed for the suffix arrays of two arbitrary strings A and B.

In this paper, we present efficient algorithms for merging the suffix arrays of two arbitrary strings A and B. The main operation of merging suffix arrays of A and B are searching all suffixes of string B in the suffix array of A. For constant alphabet, we present two merging algorithms. One algorithm uses suffix links in suffix arrays defined by Abouelhoda et al.[16] to do the main operation and the other uses backward search introduced by Ferragina and Manzini[17,18] to do the main operation. The experimental results show that merging two suffix arrays of A and B using our algorithms are about 5 times faster than constructing the suffix array of $A\#B\$$ from scratch. For integer alphabet, we present a merging algorithm without range minima query which corresponds to the LCA (lowest common ancestor) operation in the suffix trees[19].

We also present two efficient methods to compute suffix links in suffix arrays. Until now, the fastest running time for computing the suffix links has been either $O(n\log n)$or $O(n)$ using the range minima query[16] where n is the length of the string. Both of our methods runs in $O(n)$ time without range minima query: One is for constant alphabet and the other is for integer alphabet. The experimental results show that our methods for computing suffix links are about 3-4 times faster than the previous fastest methods for constant alphabet.

We introduce some notations and definitions in Section 2. In Section 3, we describe the suffix links and present efficient algorithms for computing them. In Section 4, we describe the construction of generalized suffix arrays for constant alphabets. In Section 5, we measure the running time of our algorithm and compare it with those of previous algorithms.

2 Preliminaries

Consider a string S of length n over an alphabet Σ. Let $S[i]$ for $1\le i\le n$ denote the ith symbol of string S. We assume that $S[n]$ is a special symbol # or \$ which are lexicographically larger than any other symbol in Σ. (# is larger than \$, i.e., \$<#.)

The *suffix array* $\mathsf{pos}[1..n]$ of S is basically a sorted list of all the suffixes of S. However, suffixes themselves are too heavy to be stored and thus only the starting positions of the suffixes are stored in the **pos** array. Figure 1 shows an example of the **pos** array of *accttacgacgaccttcca#*. We will consider, in this paper, the starting position of a suffix as the suffix itself.

The *lcp array* $\mathsf{lcp}[1..n]$ of S is an array that stores the lengths of the longest common prefix of two adjacent suffixes in the **pos** array. We store in $\mathsf{lcp}[i]$, $2\le i\le n$, the length of the longest common prefix of $\mathsf{pos}[i-1]$ and $\mathsf{pos}[i]$. We store 0 in $\mathsf{lcp}[1]$. For example, in Figure 1, $\mathsf{lcp}[3]=2$ because the length of the longest common prefix of $\mathsf{pos}[2]$ and $\mathsf{pos}[3]$ is 2.

The *lcp-interval* of the suffix array of S, corresponding to the internal node in the suffix tree of S, is defined as follows[12].

Definition 1. Interval $[i..j]$, $0\le i<j\le n$, is an lcp-interval of lcp-value l (l-interval), if

1. $\mathsf{lcp}[i]<l$,
2. $\mathsf{lcp}[k]\ge l$ for all $i+1\le k\le j$,
3. $\mathsf{lcp}[k]=l$ for some $i+1\le k\le j$, and
4. $\mathsf{lcp}[j+1]<l$.

For example, in Figure 1, interval $[1..4]$ is a 2-internal because $\mathsf{lcp}[1]<2$, $\mathsf{lcp}[k]\ge 2$ for all $2\le k\le 4$, $\mathsf{lcp}[3]=2$, and $\mathsf{lcp}[5]<2$. *The prefix* of an lcp-interval $[i..j]$ is the longest common prefix of the suffixes in $\mathsf{pos}[i..j]$. Figure 1 shows the one-to-one correspondence between the lcp-

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
a given string	*a*	*c*	*c*	*t*	*t*	*a*	*c*	*g*	*a*	*c*	*g*	*a*	*c*	*c*	*t*	*t*	*c*	*c*	*a*	#
pos	1	12	9	6	19	18	17	2	13	10	7	3	14	11	8	5	16	4	15	20
lcp	0	5	2	5	1	0	1	2	4	1	4	1	3	0	3	0	1	1	2	0
corresponding suffixes	*a*	*a*	*a*	*a*	*a*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*g*	*g*	*t*	*t*	*t*	*t*	#
	c	*c*	*c*	*c*	#	*a*	*c*	*c*	*c*	*g*	*g*	*t*	*t*	*a*	*a*	*a*	*c*	*t*	*t*	
	c	*c*	*g*	*g*			*a*	*t*	*t*	*a*	*a*	*t*	*t*	*c*	*c*			*a*	*c*	
	t	*t*	*a*	*a*				*t*	*t*	*c*	*c*	*a*	*c*	*c*	*g*					
	t	*t*	*c*	*c*				*a*	*c*	*c*	*g*									
	a	*c*	*c*	*g*																
up			2		3	5				8		11		7		15				17
down	5		4			7	8	9		11		13		15		17		19		
next index	6					14	10			12				16		20	18			
cldtab	6	2	4	3	5	14	10	9	8	12	11	13	7	16	15	20	18	19	17	
suflink start		8	6	10	1	1	1	6	12	1	14	1	18	1	1	1	1	1	16	1
suflink end		9	13	11	20	20	20	13	13	20	15	20	19	20	4	20	20	20	19	20

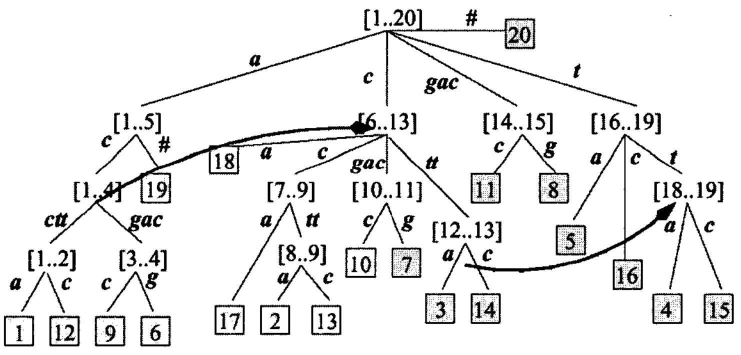


Fig. 1 The suffix tree and array of *accttacgacgaccttcca#*

intervals in the suffix array and the internal nodes in the suffix tree. The parent-child relationship between the lcp-intervals are the same as that between the corresponding nodes in suffix trees. That is, an lcp-interval $[i..j]$ is a *child interval* of another lcp-interval $[k..l]$ if the corresponding node of the lcp-interval $[i..j]$ is a child node of the corresponding node of the lcp-interval $[k..l]$. An lcp-interval $[i..j]$ is the *parent interval* of an lcp-interval $[k..l]$ if the lcp-interval $[k..l]$ is a child interval of the lcp-interval $[i..j]$. For example, in Figure 1, lcp-interval [1..4] is a child interval of lcp-interval [1..5] and lcp-interval [1..5] is the parent interval of lcp-interval [1..4].

The child table **cldtab**, introduced by Abouelhoda et al.[16], is the incorporation of three conceptual tables **up**, **down**, and **nextlIndex**. It stores some information about the parent-child relationship between lcp-intervals. The child table simulates the suffix tree where each node maintains its child pointers by a linked list and has no parent pointer. We can find the first index of the second child interval of a given lcp-interval $[i..j]$ in $O(1)$ time using **up**$[j+1]$ and **down**$[i]$. If $i <$ **up**$[j+1] \leq j$, **up**$[j+1]$ stores the first index of the second child interval. Otherwise, **down**$[i]$ stores it. For example, consider lcp-interval [18..19] in Figure 1. We first check **up**[20] but it stores 17 which is smaller than 18. Then, we check **down**[18]. It stores 19 which is the first index of the second child interval [19..19] of [18..19]. We can find the next sibling of lcp-interval $[i..j]$ in $O(1)$ time using **nextlIndex**$[i]$. The **nextlIndex**[6] stores 14 which is the first index of the next sibling [14..15] of [6..13]. The enhanced suffix array is composed of the **pos** array, the **lcp** array, and the **cldtab**. With this enhanced suffix array, we can efficiently solve all problems that are usually solved by top-down traversal of the suffix trees.

3 Computing suffix links

Suffix links introduced by Abouelhoda et al.[16] are defined on lcp-intervals. Let a and α denote a symbol and a string, respectively. The suffix link of a k-interval $[u..v]$ whose prefix is $a\alpha$ is the $(k-1)$-interval $[p..q]$ whose prefix is α. For example, in Figure 1, the suffix link of 3-interval [12..13] whose prefix is *ctt* is 2-interval [18..19] whose prefix is *tt*, and the suffix link of 2-interval [1..4] whose prefix *ac* is 1-interval [6..13] whose prefix is *a*.

We store the suffix links in table **slink**$[1..n]$. Let the suffix link of an lcp-interval $[u..v]$ be lcp-interval $[p..q]$. When we store the suffix link of the lcp-interval $[u..v]$, we only store the first and the last indices of the lcp-interval $[p..q]$, i.e., p and q. We store them in **slink**$[x]$ where x is the first index of the second child interval of the lcp-interval $[u..v]$. For example, we store the suffix link of lcp-interval [1..4], which is [6..13], in **slink**[3] because 3 is the first index of the lcp-interval [3..4] which is the second child interval of the lcp-interval [1..4]. Since every lcp-interval has two or more children, every lcp-interval can store its suffix link in a unique place. Consider the time required to access the suffix link of an lcp-interval. If we have the child table, we can find the first index of the second child interval in $O(1)$ time and thus we can access the suffix link of the lcp-interval in $O(1)$ time.

We describe how to compute the suffix links in $O(n)$ time without range minima query. We first describe for constant alphabet and then for integer alphabet. If the size of alphabet is constant, we can compute all the suffix links of the lcp-intervals by performing a preorder depth-first traversal on the enhanced suffix array as McCreight did in constructing suffix trees[2]. During the traversal, we perform the followings every time we encounter an lcp-interval $[i..j]$ of an lcp-value l.

1. Let $[u..v]$ be the parent interval of lcp-interval $[i..j]$ and k be the length of prefix of the parent interval $[u..v]$. Let $[x..y]$ be the suffix link of the parent interval $[u..v]$. The suffix link $[x..y]$ was already computed because we compute the suffix links by performing the preorder traversal. Thus, we can find the suffix link $[x..y]$ in $O(1)$ time.
2. In order to find the lcp-interval of lcp-value $l-1$, we traverse down the enhanced suffix array from lcp-interval $[x..y]$ using the substring $S[i+k..i+l-2]$.

The analysis of running time of this algorithm is very similar to that of McCreight's algorithm for suffix trees and thus we get the following lemma.

Lemma 1. We can compute all the suffix links in $O(n|\Sigma|)$ time for constant alphabet.

However, if we are to compute the suffix links for integer alphabets, this approach takes $O(n^2)$ time. We describe how to compute the suffix links for integer alphabets in $O(n)$ time. Since the suffix links of 1-intervals is trivially the 0-interval $[1..n]$, we only consider the suffix links of k-intervals, $k\geq 2$. We start with presenting a key property that enables one to compute the suffix links in $O(n)$ time. Let $[u..v]$ be a k-interval whose prefix is $a\alpha$. Consider the interval $[\Psi[u]..\Psi[v]]$ where $\Psi[i]=\text{pos}^{-1}[\text{pos}[i]+1]$. From the definition of Ψ, all the suffixes in **pos** $[\Psi[u]..\Psi[v]]$ has the prefix α. We call this interval the *Ψ-interval* of $[u..v]$. (The Ψ-interval of $[u..v]$ may not be an lcp-interval but it does not matter.) From the fact that all the suffixes in **pos**$[\Psi[u]..\Psi[v]]$ has the prefix α and all the suffixes whose prefixes are α should be stored in **pos**$[p..q]$ where $[p..q]$ is the suffix link of $[u..v]$, we get the following lemma.

Lemma 2. The suffix link $[p..q]$ of an lcp-interval $[u..v]$ contains the Ψ-interval of $[u..v]$, i.e., $p\leq\Psi[u]<\Psi[v]\leq q$.

By this lemma, finding the suffix link of a k-interval $[u..v]$ is finding the $(k-1)$-interval $[i..j]$ containing the Ψ-interval of $[u..v]$. (Note that only one $(k-1)$-interval contains the Ψ-interval of $[u..v]$ because no two $(k-1)$-intervals overlap.) For example, in Figure 2, the Ψ-interval

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
pos	**1**	**12**	**9**	**6**	**19**	**18**	**17**	**2**	**13**	**10**	**7**	**3**	**14**	**11**	**8**	**5**	**16**	**4**	**15**	**20**

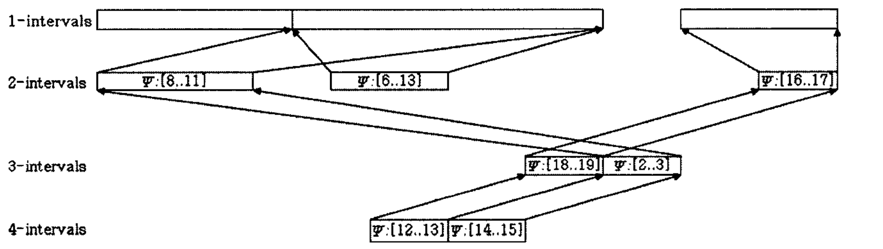


Fig. 2 The relationship between (k-1)-intervals and Ψ-intervals of k-interval.

of 2-interval [18..19] is [16..17]. The suffix link of [18..19] is [16..19] because 1-interval [16..19] contains [16..17].

Computing the suffix links consists of three steps. In these steps, we maintain two multiple lists $L[i]$ and $M[i]$, $1 \le i < n$. We use $L[i]$ to store i-intervals and $M[i]$ to store the Ψ-intervals of i-intervals.

Step 1. Compute L: We perform an inorder traversal on the enhanced suffix array. If we encounter a k-interval $[u..v]$, we insert $[u..v]$ into $L[k]$. Note that the k-intervals in $L[k]$ is sorted in the increasing order of the first indices of the k-intervals.

Step 2. Compute M: For each lcp-interval $[u..v]$ in $L[k]$, $2 \le k \le n-1$, we compute the Ψ-interval of $[u..v]$ and insert it into $M[k]$. We sort the Ψ-intervals in $M[k]$ in the increasing order of the first indices of the intervals for every $2 \le k \le n-1$. This can be done in $O(n)$ time by bucket sorting of size $n-1$ using backward pointers[20,21].

Step 3. For the k-intervals in $L[k]$, $2 \le k \le n-1$, we compute the suffix links as follows. For every Ψ-interval $[\Psi[u]..\Psi[v]]$ in $M[k]$, we find the $(k-1)$-interval $[p..q]$in $L[k-1]$ such that $p \le \Psi[u] < \Psi[v] \le q$. Then, we set the suffix link of k-interval $[u..v]$ as $(k-1)$-interval $[p..q]$. Since the intervals in $M[k]$ and $L[k-1]$ are sorted in the increasing order of the first index of the intervals, we can find all the suffix links of k-intervals in $O(c(k)+c(k-1))$ time where $c(i)$ is the number of i-intervals. Overall, step 3 can be performed in $O(n)$ time.

Since all steps are performed in $O(n)$ time, we get the following lemma.

Lemma 3. We can compute all the suffix links in $O(n)$ time for integer alphabet without range minima query.

4. Constructing generalized suffix arrays

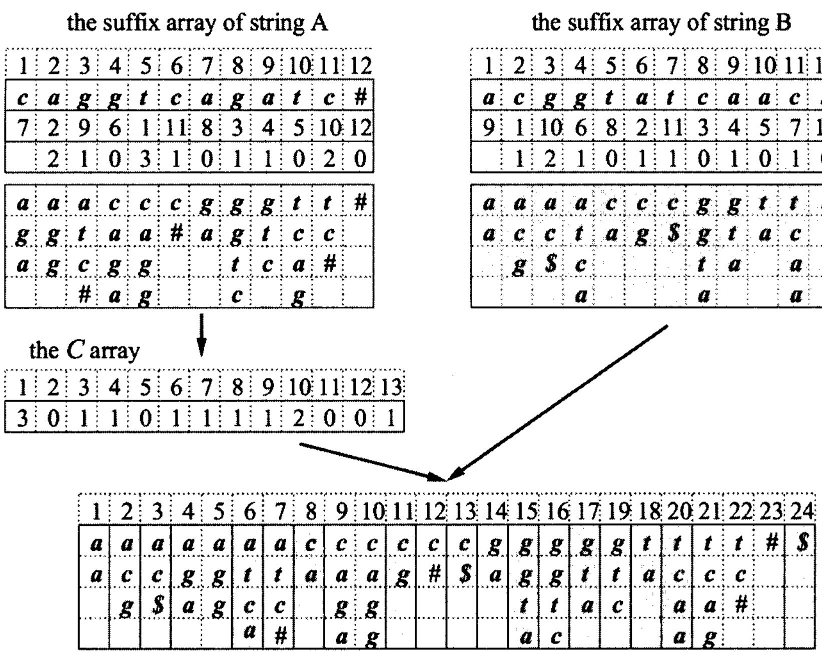
We describe how to construct the generalized suffix array of A and B when the suffix arrays of A and B are given, i.e., merging two suffix arrays of A and B. Let n and m denote the length of A and B, respectively. We first describe merging the suffix arrays using suffix links and then using backward search.

4.1 Using suffix links

Let pos_A and pos_B denote the suffix arrays of A and B, respectively. Let pos_G denote the generalized suffix array of A and B. Merging pos_A and pos_B consists of two steps and requires an additional array $C[1..n]$.

Step 1. We count the number of suffixes of B that are larger than the suffix $\mathsf{pos}_A[i-1]$ and smaller than the suffix $\mathsf{pos}_A[i]$ for all $1 \le i \le n$ and store the number in $C[i]$. To compute $C[i]$, $1 \le i \le n$, we do the following.

1. Initialize all the entries of array $C[1..n]$ to zero.
2. Construct the child table cldtab_A in $O(n)$ time.
3. Construct the suffix link table slink_A in $O(n)$ time as we described in the previous section.
4. Search the suffixes of B in pos_A, from the



the suffix array of string A

1	2	3	4	5	6	7	8	9	10	11	12
c	*a*	*g*	*g*	*t*	*c*	*a*	*g*	*a*	*t*	*c*	#
7	2	9	6	1	11	8	3	4	5	10	12
	2	1	0	3	1	0	1	1	0	2	0

a	*a*	*a*	*c*	*c*	*c*	*g*	*g*	*g*	*t*	*t*	#
g	*g*	*t*	*a*	*a*	#	*a*	*g*	*t*	*c*	*c*	
a	*g*	*c*	*g*	*g*			*t*	*c*	*a*	#	
		#	*a*	*g*			*c*		*g*		

the suffix array of string B

1	2	3	4	5	6	7	8	9	10	11	12
a	*c*	*g*	*g*	*t*	*a*	*t*	*c*	*a*	*a*	*c*	$
9	1	10	6	8	2	11	3	4	5	7	12
	1	2	1	0	1	1	0	1	0	1	0

a	*a*	*a*	*a*	*c*	*c*	*c*	*g*	*g*	*t*	*t*	$
a	*c*	*c*	*t*	*a*	*g*	$	*g*	*t*	*a*	*c*	
	g	$	*c*				*t*	*a*		*a*	
			a				*a*			*a*	

the *C* array

1	2	3	4	5	6	7	8	9	10	11	12	13
3	0	1	1	0	1	1	1	1	2	0	0	1

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	19	18	20	21	22	23	24
a	*a*	*a*	*a*	*a*	*a*	*a*	*c*	*c*	*c*	*c*	*c*	*c*	*g*	*g*	*g*	*g*	*g*	*t*	*t*	*t*	*t*	#	$
a	*c*	*c*	*g*	*g*	*t*	*t*	*a*	*a*	*a*	*g*	#	$	*a*	*g*	*g*	*t*	*t*	*a*	*c*	*c*	*c*		
	g	$	*a*	*g*	*c*	*c*		*g*	*g*					*t*	*t*	*a*	*c*		*a*	*a*	#		
					a	#		*a*	*g*					*a*	*c*				*a*	*g*			

the suffix array of string A#B$

Fig. 3 Merging the suffix arrays of A and B

longest to the shortest. This can be done in $O(m|\Sigma|)$ time using tables cldtab_A and slink_A. During the search, we increment $C[i]$ if a suffix of B is larger than $\mathsf{pos}_A[i-1]$ and smaller than $\mathsf{pos}_A[i]$.

Step 2. We store the suffixes in pos_A and pos_B into pos_G using array C. Let p_i, $1 \le i \le n$, denote prefix sum $C[1]+\cdots+C[i]$. We should store the suffix $\mathsf{pos}_A[i]$ of A into $\mathsf{pos}_G[i+p_i]$ because p_i suffixes of B are smaller than $\mathsf{pos}_A[i]$. We should store the suffixes $\mathsf{pos}_B[p_i+1..p_{i+1}]$ of B into $\mathsf{pos}_G[i+p_i+1..i+p_{i+1}]$ because i suffixes of A are smaller than the suffix $\mathsf{pos}_B[j]$, $p_i+1 \le j \le p_{i+1}$. To store the suffixes of A and B into pos_G in $O(n+m)$ time, we do the following: We store the suffixes into pos_G from the smallest to the largest. We first store $C[1]$ smallest suffixes of B into pos_G and then we store the suffix $\mathsf{pos}_A[1]$ of A into pos_G. Then, we store next $C[2]$ smallest suffixes of B then store the suffix $\mathsf{pos}_A[2]$ of A. We repeat this procedure until all the suffixes of A and B are stored in pos_G. Consider the example in Figure 3. Since $C[1]=3$, we store the three smallest suffix $\mathsf{pos}_B[1..3]$ of B into $\mathsf{pos}_G[1..3]$ and store the suffix $\mathsf{pos}_A[1]$ of A into $\mathsf{pos}_G[4]$. Since $C[2]=0$, we store no suffixes of B and then store the suffix $\mathsf{pos}_A[2]$ of A into $\mathsf{pos}_G[5]$.

Consider the time complexity of this merging algorithm. Since the step 1 takes $O(n+m|\Sigma|)$ time and step 2 takes $O(n+m)$ time, Merging pos_A and pos_B takes $O(n+m|\Sigma|)$ time. Hence, we get the following lemma.

Lemma 4. We can merge two suffix arrays of A and B in $O(n+m|\Sigma|)$ time with suffix links.

4.2 Using backward search

The backward search was introduced by Ferragina and Manzini[17,18]. They used it to develop an opportunistic data structures that uses Burrow-Wheeler transformation[22]. The backward search has been used to search patterns in compressed suffix arrays[23-25] which are suggested by Grossi and Vitter[26]. Hon et al.[15] introduced the backward search to merging the succinctly represented odd and even arrays.

Now, we consider merging pos_A and pos_B using backward search. Merging using backward search is the same as merging using suffix links except computing the array C in step 1. Thus, we only describe how to compute array C.

1. Initialize all the entries of array $C[1..n]$ to zero.
2. Generate a data structure for the backward search in pos_A in $O(n)$ time.
3. Search the suffixes of B in pos_A, from the shortest to the longest: This can be done in $O(m \cdot t_{BS})$ time where t_{BS} is the time to search a suffix of B using the backward search. During the search, we increment $C[i]$ if a suffix of B is larger than $pos_A[i-1]$ and smaller than $pos_A[i]$.

Consider the time complexity of this merging algorithm. Since the backward search in step 3 takes $O(m \cdot t_{BS})$ time and the other parts of the merging step take $O(n)$ time, this merging algorithm takes $O(n+m \cdot t_{BS})$ time. Hence, we get the following lemma.

Lemma 5. We can merge two suffix arrays of A and B in $O(n+m \cdot t_{BS})$ time using backward search.

Sim et al.[13] developed a data structure that makes the backward search fast in the suffix arrays. Using their data structure, we can search a suffix of B in $O(\log|\Sigma|)$ time, i.e., $t_{BS} = \log|\Sigma|$.

5. Experimental results

We measure the running time of constructing the generalized suffix array for two strings of the

	Build	Merge using suffix links				Merge using backward search			
length	A#B$	DS	search	total	ratio(%)	DS	search	total	ratio(%)
	$\lvert\Sigma\rvert = 2$								
1M	4.8	0.3	0.8	1.1	**22.8**	0.3	1.0	1.3	26.8
5M	25.3	1.2	4.6	5.8	**22.9**	1.7	5.6	7.3	28.9
10M	51.4	2.8	10.7	13.5	**26.2**	3.3	12.2	15.5	30.1
30M	164.8	11.3	38.2	49.5	**30.0**	10.1	40.7	50.8	30.9
	$\lvert\Sigma\rvert = 4$								
1M	6.0	0.4	0.9	1.3	**20.5**	0.3	1.0	1.3	21.3
5M	30.7	1.4	4.5	5.9	**19.0**	1.5	5.9	7.4	23.9
10M	61.5	2.6	10.3	12.9	**20.9**	3.1	12.4	15.5	25.2
30M	291.7	11.0	38.0	49.0	**16.8**	9.3	41.5	50.8	17.4
	$\lvert\Sigma\rvert = 64$								
1M	3.8	0.9	5.1	6.0	157.9	0.2	0.8	1.0	**26.5**
5M	40.4	4.2	11.8	16.0	39.7	1.0	6.0	7.0	**17.6**
10M	82.8	7.8	11.7	19.5	23.5	2.2	12.5	14.7	**17.8**
30M	257.7	14.8	34.6	49.4	**19.2**	7.2	43.5	50.7	19.7
	$\lvert\Sigma\rvert = 128$								
1M	4.0	1.1	20.2	21.3	526.2	0.2	0.6	0.8	**20.4**
5M	42.8	8.9	53.4	62.3	145.3	0.9	4.4	5.3	**12.4**
10M	87.8	21.9	45.5	67.4	76.7	2.0	12.0	14.0	**16.0**
30M	269.6	43.6	37.2	80.8	30.0	6.9	42.1	49.0	**18.2**

Fig. 4. The experimental results for constructing the generalized suffix arrays. We used Kärkkäinen and Sanders's algorithm for constructing the generalized suffix arrays from scratch and we used Sim et al.'s[13] data structure for the backward search. We measured the running time in second on the 2.8Ghz Pentium IV with 2GB main memory.

same length using our merging algorithms and compare it with constructing the generalized suffix array from scratch. We generated different kinds of random strings which are differ in lengths (1M, 5M, 10M, and 30M) and in the sizes of alphabets (2, 4, 64, and 128) from which they are drawn. Figure 4 shows that constructing the generalized suffix array by merging two suffix arrays using suffix links and backward search is about 5 times faster than constructing the generalized suffix array from scratch. Figure 4 also shows that merging using suffix links is faster when the size of alphabet is small (2 or 4), and merging using backward search is faster when the size of alphabet is quite large (64 or 128). This result is consistent with the running time analysis of the merging algorithms. (Merging using suffix links runs in $O(n+m|\Sigma|)$ time due to Lemma 4 and merging using backward search runs in $O(n+m\log|\Sigma|)$ time due to Lemma 5 and Sim et al.)

We made experiments on merging the suffix arrays of two strings of different lengths. We can merge the suffix arrays either by searching the shorter string in the suffix array of the longer string or by searching the longer string in the suffix array of the shorter string. Figure 5 shows that the former is faster. For example, when we merge two strings of lengths 10M and 2M for $|\Sigma|=4$, searching the string of length 10M in the suffix array of the string of length 2M takes 9.0 seconds and searching the string of length 2M in the suffix array of the string of length 10M takes 4.6 seconds. This result is consistent with the result that searching takes more time than constructing the suffix links shown in Figure 4.

In addition, we measure the running time of our method to compute suffix links and compare it with the two previous methods suggested by Abouelhoda et al.[16]. Two previous methods are denoted by '$O(n\log n)$' and '$O(n)$ with RMQ' in Figure 6. We generated different kinds of random strings which are differ in lengths (1M, 5M, and, 10M) and in the sizes of alphabets (2, 4, and 8) from which they are drawn. Figure 6 shows that our method compute the suffix links about 3-4 times faster

string A	string B	Merge using suffix links			Merge using backward search		
length(n)	length(m)	$\|\Sigma\|=2$	$\|\Sigma\|=4$	$\|\Sigma\|=128$	$\|\Sigma\|=2$	$\|\Sigma\|=4$	$\|\Sigma\|=128$
2M	4M	3.9	3.9	85.1	5.0	5.5	3.3
4M	2M	2.7	2.8	36.7	3.9	3.8	2.8
2M	6M	5.7	5.6	117.9	7.3	7.5	4.7
6M	2M	3.5	3.5	27.0	4.8	4.7	3.6
2M	10M	9.1	9.0	197.6	11.7	12.2	7.8
10M	2M	4.9	4.6	29.1	65.2	12.7	5.0

Fig. 5 The experimental results for merging the suffix arrays of two strings of different lengths.

alphabet	$\|\Sigma\|=2$			$\|\Sigma\|=4$			$\|\Sigma\|=8$		
length	1M	5M	10M	1M	5M	10M	1M	5M	10M
$O(n\log n)$	1.11	5.76	12.42	0.93	5.57	12.16	0.70	5.32	11.98
$O(n)$ with RMQ	2.31	16.51	37.51	2.15	15.11	36.56	1.97	14.08	33.92
Ours	0.27	1.29	2.73	0.33	1.35	2.53	0.42	1.62	2.96
$O(n\log n)$ /Ours	4.1	4.5	4.6	2.8	4.1	4.8	1.7	3.3	4.0
$O(n)$ with RMQ/Ours	8.5	12.8	13.8	6.5	11.2	14.5	4.7	8.7	11.5

Fig. 6 The experimental results for computing the suffix links.

than the previous fastest method.

6. Conclusion

We presented two fast algorithms for merging two suffix arrays of A and B when the suffix arrays of A and B are given. One uses suffix links and the other uses backward search method. The experimental results show that merging algorithm that uses suffix links is faster than the other algorithm when the alphabet size is small.

We also presented two efficient algorithms for computing suffix links in our data structure. Our algorithms can run in linear time without range minima query. It is the first practical linear time algorithm for computing suffix links because the range minima query operation that is used in the previous linear time algorithm is theoretical. We can find the longest common substring and compute matching statistics fast with suffix links.

References

[1] D. Gusfield, Algorithms on Strings, Trees, and Sequences, *Cambridge Univ. Press,* 1997.

[2] E.M. McCreight, A space-economical suffix tree construction algorithm, *J. Assoc. Comput.* Vol 23, pp. 262-272, 1976.

[3] P. Weiner, Linear pattern matching algorithms, *Proc. 14th IEEE Symp. Switching and Automata Theory,* pp. 1-11, 1973.

[4] M. Farach, Optimal suffix tree construction with large alphabets, *IEEE Symp. Found. Computer Science*, pp. 137-143, 1997.

[5] M. Farach-Colton, P. Ferragina and S. Muthukrishnan, On the sorting-complexity of suffix tree construction, *J. Assoc. Comput. Mach.,* vol 47, pp. 987-1011, 2000.

[6] E. Ukkonen, On-line construction of suffix trees, *Algorithmica*, vol. 14, pp. 249-260, 1995.

[7] U. Manber and G. Myers, Suffix arrays: A new method for on-line string searches, *SIAM J. Computing*, vol. 22, pp. 935-938, 1993.

[8] G. Gonnet, R. Baeza-Yates, and T. Snider, New indices for text: Pat trees and pat arrays. In W. B. Frakes and R. A. Baeza-Yates, editors, Information Retrieval: Data Structures & Algorithms, *Prentice Hall*, pp. 66-82, 1992.

[9] J. Kärkkäinen and P. Sanders, Simpler linear work suffix array construction, *Int. Colloq. Automata Languages and Programming*, pp. 943-955, 2003.

[10] D. K. Kim, J. S. Sim, H. Park and K. Park, Linear-time construction of suffix arrays, *Symp. Combinatorial Pattern Matching*, pp. 186-199, 2003.

[11] P. Ko and S. Aluru, Space-efficient linear time construction of suffix arrays, *Symp. Combinatorial Pattern Matching*, pp. 200-210, 2003.

[12] M. Abouelhoda, E. Ohlebusch, and S. Kurtz, Optimal exact string matching based on suffix arrays, *Symp. on String Processing and Information Retrieval*, pp. 31-43, 2002.

[13] J. S. Sim, D. K. Kim, H. Park and K. Park, Linear-time search in suffix arrays, *Australasian Workshop on Combinatorial Algorithms*, pp. 139-146, 2003.

[14] M. T. Chen and J. Seiferas, Efficient and elegant subword tree construction, In A. Apostolico and Z. Galil, editors, Combinatorial Algorithms on Words, NATO ASI Series F: Computer and System Sciences, 1985.

[15] W.K. Hon, K. Sadakane and W.K. Sung, Breaking a time-and-space barrier in constructing full-text indices, *IEEE Symp. Found. Computer Science*, pp. 251-260, 2003.

[16] M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch, Replacing suffix trees with enhanced suffix arrays, *J. of Discrete Algorithms*, pp. 53-86, 2004.

[17] P. Ferragina and G. Manzini, Opportunistic data structures with applications, *IEEE Symp. Found. Computer Science*, pp. 390-398, 2001.

[18] P. Ferragina and G. Manzini, An experimental study of an opportunistic index, *ACM-SIAM Symp. on Discrete Algorithms*, pp. 269-278, 2001.

[19] M. Bender and M. Farach-Colton, The LCA Problem Revisited, *In Proceedings of LATIN 2000*, LNCS vol 1776, pp. 88-94, 2000.

[20] A. Aho, J. Hopcroft, J. Ullman, Data Structures and Algorithms, *Addison-Wesley*, 1983.

[21] D. K. Kim and K. Park, Linear-time construction of two-dimensional suffix trees, *Int. Colloq. on Automata, Languages and Programming,* pp. 463-472, 1999.

[22] M. Burrows and D. Wheeler, A block sorting lossless data compression algorithm, Technical Report 124, Digital Equipment Corporation, 1994.

[23] R. Grossi, A. Gupta and J.S. Vitter, When indexing equals compression: Experiments with compressing suffix arrays and applications, *ACM-SIAM Symp. on Discrete Algorithms*, 2004.

[24] K. Sadakane, Compressed text databases with efficient query algorithms based on the compressed suffix array, *Int. Symp. Algorithms and Computation*, pp. 410-421, 2000.

[25] K. Sadakane, Succinct representations of lcp Information and improvements in the compressed suffix arrays, *CM-SIAM Symp. on Discrete Algorithms*, pp. 225-232, 2002.

[26] R. Grossi and J.S. Vitter, Compressed suffix

arrays and suffix trees with applications to text indexing and string matching, *ACM Symp. Theory of Computing*, pp. 397-406, 2000.

전 정 은
2003년 부산대학교 컴퓨터공학과 학사
2005년 부산대학교 컴퓨터공학과 석사
관심분야는 알고리즘, Bioinformatics

박 희 진
1994년 서울대학교 컴퓨터공학과 학사
1996년 서울대학교 컴퓨터공학과 석사
2001년 서울대학교 컴퓨터공학과 박사
2001년~2003년 서울대학교 컴퓨터연구소 전문연구원. 2003년 이화여자대학교 컴퓨터학과 BK연구교수. 2003년~현재 한양대학교 정보통신대학 전임강사. 관심분야는 알고리즘, 컴퓨터 보안 시스템, Bioinformatics

김 동 규
1992년 서울대학교 컴퓨터공학과 학사
1994년 서울대학교 컴퓨터공학과 석사
1999년 서울대학교 컴퓨터공학과 박사
1999년~현재 부산대학교 컴퓨터공학과 조교수. 관심분야는 암호학 및 알고리즘, 컴퓨터 보안 시스템, Bioinformatics