접미사 배열을 이용한 시간과 공간 효율적인 검색
(Time and Space Efficient Search with Suffix Arrays)

최 용 욱 † 심 정 섭 †† 박 근 수 †††
(Yong Wook Choi) (Jeong Seop Sim) (Kunsoo Park)

요 약 길이가 n인 알파벳 Σ상의 텍스트 T에서 패턴 P를 효율적으로 검색하기 위해 접미사 트리와 접미사 배열이 널리 쓰이고 있다. 접미사 배열이 접미사 트리보다 더 적은 공간을 사용하기 때문에 텍스트의 길이가 긴 경우에는 접미사 배열이 더 선호되고 있다. 최근에는 접미사 배열을 이용한 $O(|P| \cdot |\Sigma|)$ 시간과 $O(|P| \cdot \log|\Sigma|)$ 시간 검색 알고리즘들이 개발되었다.

본 논문에서는 접미사 배열을 이용한 시간과 공간 효율적인 알고리즘들을 제시한다. 하나의 알고리즘은 $O(|P| \cdot |\Sigma|)$ 비트 공간을 사용하여 $O(|P|)$ 시간에 수행되고, 다른 하나는 $O(n \cdot \log|\Sigma| + |\Sigma| \cdot n\log\log n/\log n)$ 비트 공간을 사용하여 $O(|P| \cdot \log|\Sigma|)$ 시간에 수행되는데, 두 번째 알고리즘은 보다 효율적인 공간을 사용하면서 여전히 빠른 알고리즘이다. 본 논문이 제시하는 알고리즘들이 시간과 공간에 있어 기존의 알고리즘들보다 더 효율적인 알고리즘들임을 실험을 통해 보여주고 있다.

키워드 : 접미사 배열, 접미사 트리, 패턴 검색

Abstract To search efficiently a text T of length n for a pattern P over an alphabet Σ, suffix trees and suffix arrays are widely used. In case of a large text, suffix arrays are preferred to suffix trees because suffix arrays take less space than suffix trees. Recently, $O(|P| \cdot |\Sigma|)$-time and $O(|P| \cdot \log|\Sigma|)$-time search algorithms in suffix arrays were developed.

In this paper we present time and space efficient search algorithms in suffix arrays. One algorithm runs in $O(|P|)$ time using $O(n \cdot |\Sigma|)$-bits space, and the other runs in $O(|P| \cdot \log|\Sigma|)$ time using $O(n\log|\Sigma| + |\Sigma| \cdot n\log\log n/\log n)$-bits space, which is more space efficient and still fast. Experiments show that our algorithms are efficient in both time and space when compared to previous algorithms.

Key words : suffix array, suffix tree, pattern search

1. Introduction

Suffix trees and suffix arrays are two well-known index data structures for searching patterns. The suffix tree due to McCreight[1] is a compacted trie of all the suffixes of a string T. It was designed as a simplified version of Weiner's position tree[2]. The suffix array due to Manber and Myers [3] and independently due to Gonnet et al. [4] is basically a sorted list of all the suffixes of a string T. Recently, there arise needs of searching patterns from a large amount of data such as DNA sequences in computational biology. Thus suffix arrays are preferred to suffix trees because suffix arrays consume less space than suffix trees[5-12].

There have been vigorous works on searching patterns with suffix arrays. Ferragina and Manzini [13] developed an $O(|P| + z\log^{1+\epsilon} n)$-time algorithm in a file compressed by the Burrows-Wheeler compression, where $0<\epsilon<1$ is a constant and z is the number of occurrences of P in a string of length n. Recently, Abouelhoda et al. [14] and Sim et al. [15] developed linear-time search algorithms in suffix arrays when the alphabet size is a constant. The former simulates suffix trees in suffix arrays and the latter uses backward search

† 비 회 원 : Purdue University Dept. of Computer Sciences
ywchoi@pundue.edu
†† 종신회원 : 인하대학교 컴퓨터공학부 교수
jssim@inha.ac.kr
††† 종신회원 : 서울대학교 컴퓨터공학부 교수
kpark@theory.snu.ac.kr
논문접수 : 2004년 8월 16일
심사완료 : 2005년 1월 20일

developed by Ferragina and Manzini in the context of compressed pattern matching[13]. In general, however, these algorithms take time $O(|P| \cdot |\Sigma|)$ and $O(|P| \cdot \log|\Sigma|)$, respectively.

In this paper we present time and space efficient search algorithms in suffix arrays. We use backward search in[13,15], but focus on efficient implementations based on suffix arrays. One implementation runs in $O(|P|)$ time using $O(n \cdot |\Sigma|)$ -bits space, and the other runs in $O(|P| \cdot \log|\Sigma|)$ time using $O(n\log|\Sigma| + |\Sigma| \cdot n\log\log n/\log n)$-bits space. We conducted experiments when $|\Sigma| = 4$ and 20. It is shown that our implementations are efficient in both time and space when compared to Abouelhoda et al.[14] and Sim et al.[15].

2. Preliminaries

Consider a string T of length n over an alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}\}$, where σ_i is lexicographically smaller than σ_j if $i<j$. For $1 \le i \le n$, $T[i]$ denotes the symbol at position i of string T and $T[i..j]$ the substring of T that starts at position i and ends at position j. S_i denotes the suffix of T that starts at position i, that is, $T[i..n]$. We assume that $T[n]$ is a special symbol # which is lexicographically smaller than any other symbol in Σ. The suffix array A_T is an array of integers i that represent suffixes S_i of T. The integers are sorted in lexicographic order of the corresponding suffixes. See Figure 1(a) for an example. In the rest of this paper, "i-th suffix" will mean the i-th suffix in lexicographic order. For example, the 10-th suffix in Figure 1(a) is *baa#* and conversely the index of suffix *baa#* is 10.

We define an array $Prev[1..n]$ as follows:

$$Prev[i] = \begin{cases} T[A_T[i]-1] & \text{if } A_T[i] \neq 1 \\ T[n] & \text{if } A_T[i] = 1. \end{cases}$$

In other words, $Prev[i]$ is the previous symbol of $S_{A_T[i]}$ in T if we assume $T[0] = \#$. See Figure 1(a) for an example. Burrows and Wheeler[16] introduced this array, which was named string L, the last column of the matrix M in their paper. Ferragina and Manzini[13] developed a search algorithm using $Prev$, which is known as backward search, in the context of FM-index.

We define an array $Base$ where $Base[\sigma_i]$, $1 \le i \le |\Sigma|$, stores the number of symbols that are lexicographically smaller than σ_i in T. For convenience, we assume $Base[\sigma_{|\Sigma|+1}] = n$. Note that $Base[\sigma_i]+1$ means the position of the lexicographically smallest suffix that starts with σ_i. In Figure 1(a), $Base[\#] = 0$, $Base[a] = 1$, $Base[b] = 9$ and the 10-th suffix is the smallest one that starts with b.

We also define a function $Count(\sigma_i, j)$ for $1 \le i \le |\Sigma|$ and $0 \le j \le n$, which returns the number of occurrences of σ_i in $Prev[1..j]$. For convenience, we

i	$A_T[i]$	$Prev[i]$	$S_{A_T[i]}$
1	14	a	#
2	13	a	a#
3	12	b	aa#
4	7	b	aababaa#
5	1	#	aabbabaabaabaa#
6	10	b	abaa#
7	5	b	abaababaa#
8	8	a	ababaa#
9	2	a	abbabaababaa#
10	11	a	baa#
11	6	a	baababaa#
12	9	a	babaa#
13	4	b	babaabaabaa#
14	3	a	bbabaababaa#

(a)

i	$A_T[i]$	$Prev[i]$	$S_{A_T[i]}$
1	14	a	#
2	13	a	a#
3	12	C_3 b	aa#
4	7	C_4 b	aababaa#
5	1	#	aabbabaabaabaa#
6	10	C_6 b	abaa#
7	5	C_7 b	abaababaa#
8	8	a	ababaa#
9	2	a	abbabaababaa#
10	11	a	baa#
11	6	a	baababaa#
12	9	a	babaa#
13	4	C_{13} b	babaabaabaa#
14	3	a	bbabaababaa#

(b)

Figure 1 $A_T[i]$, $Prev[i]$, and C_i when $T = aabbabaababaa\#$

assume $Count(\sigma_i, 0) = 0$ for $1 \le i \le |\Sigma|$. For example, $Count(a,0) = 0$, $Count(\#,4) = 0$, $Count(a,10) = 5$, and $Count(b,8) = 4$ in Figure 1(a).

3. Backward search

Let P be a pattern string of length m over an alphabet Σ. Now we are considering the problem of counting the number of occurrences of pattern P in T. Since all suffixes of T are lexicographically ordered in suffix array A_T, we can find all suffixes that have P as their prefixes in a contiguous set of indices in A_T. Thus, by finding the first index i_f and the last index i_l of this set, we can count the number of occurrences of P, i.e., $i_l - i_f + 1$. In Figure 1(a), for example, a pattern $P = ba$ appears as prefixes of the 10-th to 13-th suffixes and the number of occurrences of P is 4.

Let C_i be the concatenation of $Prev[i]$ and $S_{A_T[i]}$ for $1 \le i \le n$ (In case that $S_{A_T[i]} = T$, we assume C_i is #). In Figure 1(b), $C_3 = baa\#$ because $Prev[3] = b$ and $S_{A_T[3]} = aa\#$. It is easy to see that C_i is a suffix of T and there is a one-to-one correspondence between strings C_i and all suffixes $S_{A_T[i]}$.

Lemma 1[15] Let C_i, which corresponds to $S_{A_T[i]}$, be the j-th string among the suffixes starting with a symbol $\sigma \in \Sigma$. Then $S_{A_T[i']}$ is the $(Base[\sigma] + j)$-th suffix among all the suffixes of T, i.e., $i' = Base[\sigma] + j$.

All C_i's that start with a same symbol are lexicographically ordered because the $S_{A_T[i]}$ parts of the C_i's are already ordered. For example, see Figure 1(b) for the strings that start with b. All strings C_i in $\{C_3, C_4, C_6, C_7, C_{13}\}$ are lexicographically ordered. We can also see that all C_i's correspond to the $(Base[b]+1)$-st to $(Base[b]+5)$-th suffixes (i.e., the 10-th to 14-th) in order by Lemma 1. Since these suffixes make a contiguous block, we can represent them by their first and last indices, i.e., 10 and 14.

Now we describe the backward search and analyze its time complexity. We search for P from the last symbol to the first symbol of P. For the last symbol $P[m]$ of P, we can find the indices of the suffixes that start with $P[m]$, just using array $Base$. By definition of array $Base$, the first index is $Base[P[m]]+1$ and the last index is $Base[\sigma]$, where σ is the next symbol of $P[m]$. For any k, $1 \le k < m$, assume we know all suffixes that have $P[k+1..m]$ as their prefixes, i.e., we know the set of indices of those suffixes. Let i_f and i_l be the first and last indices of this set. Now we will find the set of indices of all suffixes that have $P[k..m]$ as their prefixes. Let i_f' and i_l' be the first and last indices of this set, which we will find in this step. We can find indices i_f' and i_l' by finding only two suffixes that correspond to the first and the last C_i such that $i_f \le i \le i_l$ and $Prev[i] = P[k]$. Assume that the first C_i is the j-th string among all C_i's starting with $P[k]$. By Lemma 1, $i_f' = Base[P[k]] + j$, and by definition of function $Count$, $j = Count(P[k], i_f - 1) + 1$. Thus

$$i_f' = Base[P[k]] + Count(P[k], i_f - 1) + 1. \quad (1)$$

Similarly, assume that the last C_i is the j'-th string among all C_i's starting with $P[k]$. Then, $i_l' = Base[P[k]] + j'$, and $j' = Count(P[k], i_l)$. Thus

$$i_l' = Base[P[k]] + Count(P[k], i_l). \quad (2)$$

Figure 2 shows how to find $P = bab$ in $T = aabbabaababaa\#$ step by step. When $k=3$, the initial values of i_f and i_l are computed from array $Base$, i.e., $i_f = Base[b] + 1 = 10$, $i_l = Base[\sigma_{|\Sigma|+1}] = n = 14$. Now we have the positions where $P[3] = b$ occurs in T. When $k=2$, we find all the positions where $P[2..3] = ab$ occurs in T. By (1), $i_f' = Base[P[2]] + Count(P[2], i_f - 1) + 1 = Base[a] + Count(a,9) + 1 = 6$, and by (2), $i_l' = Base[P[2]] + Count(P[2], i_l) = Base[a] + Count(a,14) = 9$. Finally, when $k=1$, we find all the positions where $P[1..3] = bab$ occurs in T. By (1), $i_f' = Base[b] + Count(b,5) + 1 = 12$, and by (2) $i_l' = Base[b] + Count(b,9) = 13$. Hence, the number of occurrences of $P = bab$ is $i_l - i_f + 1 = 2$ and P appears as prefixes of the 12-th and 13-th suffixes.

Figure 3 shows this algorithm. The running time

	i	$A_T[i]$	$Prev[i]$	$S_{A_T[i]}$
	1	14	a	#
	2	13	a	a#
	3	12	b	aa#
	4	7	b	aababaa#
	5	1	#	aabbabaababaa#
$i_f' \rightarrow$	6	10	b	abaa#
	7	5	b	abaababaa#
	8	8	a	ababaa#
$i_l' \rightarrow$	9	2	a	abbabaababaa#
$i_f \rightarrow$	10	11	a	baa#
	11	6	a	baababaa#
	12	9	a	babaa#
	13	4	b	babaababaa#
$i_l \rightarrow$	14	3	a	bbabaababaa#

(a)

	i	$A_T[i]$	$Prev[i]$	$S_{A_T[i]}$
	1	14	a	#
	2	13	a	a#
	3	12	b	aa#
	4	7	b	aababaa#
	5	1	#	aabbabaababaa#
$i_f \rightarrow$	6	10	b	abaa#
	7	5	b	abaababaa#
	8	8	a	ababaa#
$i_l \rightarrow$	9	2	a	abbabaababaa#
	10	11	a	baa#
	11	6	a	baababaa#
$i_f' \rightarrow$	12	9	a	babaa#
$i_l' \rightarrow$	13	4	b	babaababaa#
	14	3	a	bbabaababaa#

(b)

Figure 2 Search $P=bab$ in $T=aabbabaababaa\#$

Algorithm BACKWARD($P[1..m]$)

$k = m$;
$\sigma_i = P[m]$;
$i_f = Base[\sigma_i] + 1$, $i_l = Base[\sigma_{i+1}]$;
while ($i_f \le i_l$ **and** $k \ge 2$) **do**
  $k = k - 1$;
  $\sigma = P[k]$;
  $i_f = Base[\sigma] + Count(\sigma, i_f - 1) + 1$;
  $i_l = Base[\sigma] + Count(\sigma, i_l)$;
if ($i_l < i_f$)
  return "no occurrence"
else
  return "$(i_l - i_f + 1)$ occurrences"

Figure 3 Algorithm for counting the number of occurrences of pattern P in T

i	$A_T[i]$	*bit arrays* B_a	*bit arrays* B_b	$Prev[i]$	$S_{A_T[i]}$
1	14	1	0	a	#
2	13	1	0	a	a#
3	12	0	1	b	aa#
4	7	0	1	b	aababaa#
5	1	0	0	#	aabbabaababaa#
6	10	0	1	b	abaa#
7	5	0	1	b	abaababaa#
8	8	1	0	a	ababaa#
9	2	1	0	a	abbabaababaa#
10	11	1	0	a	baa#
11	6	1	0	a	baababaa#
12	9	1	0	a	babaa#
13	4	0	1	b	babaababaa#
14	3	1	0	a	bbabaababaa#

Figure 4 Bit arrays when $T=aabbabaababaa\#$

of *BACKWARD* depends on the time complexity of computing *Count*. We will present two implementations of the *Count* function. One computes *Count* in $O(1)$ time using $O(n \cdot |\Sigma|)$-bits space and the other does in $O(\log|\Sigma|)$ time using $O(n\log|\Sigma| + |\Sigma| \cdot n\log\log n/\log n)$-bits space.

4. $O(|P|)$-time algorithm

We describe how to get a *Count* value in constant time. From array *Prev* we make a bit array $B_{\sigma_i}[1..n]$ for every $\sigma_i \in \Sigma$, which is defined as follows:

$$B_{\sigma_i}[j] = \begin{cases} 1 & \text{if } Prev[j] = \sigma_i \\ 0 & \text{otherwise.} \end{cases}$$

See Figure 4 for an example. Let $num_{\sigma_i}(j,k)$ be the number of '1's in $B_{\sigma_i}[j..k]$. Since $Count(\sigma_i, j)$ is the number of occurrences of σ_i in $Prev[1..j]$, $Count(\sigma_i, j)$ is the same as $num_{\sigma_i}(1,j)$.

We use two sub-tables and one lookup table to get $num_{\sigma_i}(1,j)$ in constant time[17]. We make sub-tables for every $\sigma_i \in \Sigma$. Consider a symbol σ_i. Let l be the word size on a RAM and L be a multiple of l. First, we conceptually divide array B_{σ_i} into blocks of length L. Then we construct the first sub-table $ST^1_{\sigma_i}$, storing the number of '1's in the first k blocks into $ST^1_{\sigma_i}[k]$. Thus, for $0 \le k \le \lfloor n/L \rfloor$,

$$ST^1_{\sigma_i}[k] = \begin{cases} 0 & \text{if } k=0 \\ num_{\sigma_i}(1, kL) & \text{if } 1 \le k \le \lfloor n/L \rfloor . \end{cases}$$

We construct the second sub-table $ST^2_{\sigma_i}$ in a

similar way. We divide the first block of length L into sub-blocks of length l. Then, we store the number of '1's in the first k sub-blocks into $ST^2_{\sigma_i}[k]$ for $k=0,1,2,\ldots,L/l-1$. We do the same for every block. All numbers calculated in each block are stored in $ST^2_{\sigma_i}$ as follows: for $0 \le k \le \lfloor n/l \rfloor$,

$$ST^2_{\sigma_i}[k] = \begin{cases} 0 & \text{if } k \text{ is a multiple of } L/l \\ num_{\sigma_i}(L \cdot \lfloor kl/L \rfloor + 1, kl) & \text{otherwise.} \end{cases}$$

The lookup table LT, which has $2^{l/2}$ entries, is a common table for all σ_i's. $LT[j]$, $0 \le j < 2^{l/2}$, stores the number of '1's in the binary representation of its index j. For example, LT [0x0001] = 1 and LT [0xffff] = 16.

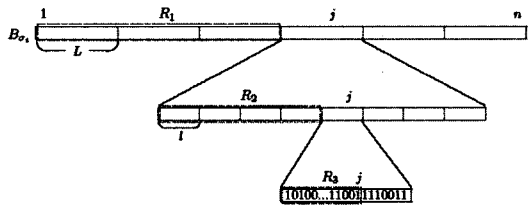

Figure 5 Computing $num_{\sigma_i}(1, j)$

We compute $Count(\sigma_i, j) = num_{\sigma_i}(1, j)$ by using array B_{σ_i} and three tables, $ST^1_{\sigma_i}$, $ST^2_{\sigma_i}$, and LT. First we conceptually divide $B_{\sigma_i}[1..j]$ into three subranges. Let $R_t (t=1,2,3)$ be the t-th subrange and N_t be the number of '1's in R_t. R_1 is a subrange of 1 to k_1L, where $k_1 = \lfloor j/L \rfloor$. R_2 is a subrange of k_1L+1 to k_2l, where $k_2 = \lfloor j/l \rfloor$. R_3 is a subrange of k_2l+1 to j. See Figure 5. By definition of num_{σ_i}, we can see $N_1 = num_{\sigma_i}(1, k_1L)$, $N_2 = num_{\sigma_i}(k_1L+1, k_2l)$, $N_3 = num_{\sigma_i}(k_2l+1, j)$, and $num_{\sigma_i}(1, j) = N_1 + N_2 + N_3$ because the range of 1 to j is the concatenation of three subranges. Now we can find $N_1 = num_{\sigma_i}(1, k_1L)$ in $ST^1_{\sigma_i}[k_1]$ and $N_2 = num_{\sigma_i}(k_1L+1, k_2l)$ in $ST^2_{\sigma_i}[k_2]$. Note that since $\lfloor k_2l/L \rfloor = k_1$, $num_{\sigma_i}(k_1L+1, k_2l) = num_{\sigma_i}(L \cdot \lfloor k_2l/L \rfloor + 1, k_2l) = ST^2_{\sigma_i}[k_2]$. We can compute $N_3 = num_{\sigma_i}(k_2l+1, j)$ by looking up table LT. We pick up the sub-block w that starts at k_2l+1. Then we remove '1's positioned after j by doing a bitwise AND operation on w. Now we can get N_3 by looking up LT twice with the upper and lower halves of w. See Figure 6.

Algorithm COUNT(σ, i)
▷ w, $mask$: binary of length l
▷ $half_l$, $half_u$: binary of length $l/2$
$N_1 = ST^1_\sigma[\lfloor i/L \rfloor]$;
$N_2 = ST^2_\sigma[\lfloor i/l \rfloor]$;
$s = \lfloor i/l \rfloor \cdot l + 1$; $e = \lfloor i/l \rfloor \cdot l + l$;
$w = B_\sigma[s..e]$;
$mask = 2^l - 2^{l-(i-\lfloor i/l \rfloor \cdot l)}$;
$w = w$ & $mask$;
$half_l = w[1..l/2]$; $half_u = w[l/2+1..l]$;
$N_3 = LT[half_l] + LT[half_u]$;
return $N_1 + N_2 + N_3$;

Figure 6 $Count(\sigma, i)$ for $O(|P|)$-time algorithm

Now we analyze the space requirement of our data structures. For every $\sigma_i \in \Sigma$, array B_{σ_i} has n entries, each of which takes 1 bit. Thus B_{σ_i} takes n bits. Since sub-table $ST^1_{\sigma_i}$ has $\lfloor n/L \rfloor + 1$ entries, each of which takes $\log n$ bits, $ST^1_{\sigma_i}$ takes about $(n \log n)/L$ bits. Similarly, sub-table $ST^2_{\sigma_i}$ has $\lfloor n/l \rfloor + 1$ entries, each of which takes $\log L$ bits, and thus it takes about $(n \log L)/l$ bits. Since there exist B_{σ_i}, $ST^1_{\sigma_i}$, and $ST^2_{\sigma_i}$ for every σ_i, $1 \le i \le |\Sigma|$, the space occupancies of B_{σ_i}'s, $ST^1_{\sigma_i}$'s, and $ST^2_{\sigma_i}$'s are about $|\Sigma| \cdot n$, $(|\Sigma| \cdot n \log n)/L$, and $(|\Sigma| \cdot n \log L)/l$ bits, respectively. Lookup table LT has $2^{l/2}$ entries, each of which takes $\log l/2$ bits, and thus it takes $2^{l/2} \cdot \log l/2$ bits. If we set $L = \Theta(\log^2 n)$ and $l = \Theta(\log n)$, the space complexities of B_{σ_i}'s, $ST^1_{\sigma_i}$'s, $ST^2_{\sigma_i}$'s, and LT are $O(n \cdot |\Sigma|)$, $O(|\Sigma| \cdot n/\log n)$, $O(|\Sigma| \cdot n \log\log n/\log n)$, and $O(\sqrt{n} \log\log n)$, respectively. Hence, the total space becomes $O(n \cdot |\Sigma|)$ bits.

We can set $l=32$ and $L=256$ in actual implementation for space-efficiency. In this case, the spaces for one entry of $ST^1_{\sigma_i}$, $ST^2_{\sigma_i}$, and LT are 4, 1, and 1 bytes, respectively. Thus the total space requirement becomes about $0.17 \cdot |\Sigma| \cdot n$ bytes. Compared with[14] and[15], which take about $2n$ and $6n$ bytes respectively, this is competitive when $|\Sigma|$ is small.

5. $O(|P| \cdot \log|\Sigma|)$-time algorithm

Now we describe a more space-efficient algorithm. In our previous algorithm, arrays B_{σ_i}, which take $n \cdot |\Sigma|$ bits, occupy most of our total space. We can reduce this space to $n\log|\Sigma|$ bits at the cost of an increase in running time. Instead of making arrays $B_{\sigma_1}, B_{\sigma_2}, \ldots, B_{\sigma_{|\Sigma|}}$, we make compacted bit arrays $CB_1, CB_2, \ldots, CB_{\log|\Sigma|}$ from array *Prev* as follows. Note that an element of *Prev* is a symbol over an alphabet Σ and thus it takes $\log|\Sigma|$ bits. We extract the most significant bit of $Prev[i]$ and store it into $CB_1[i]$ for $1 \le i \le n$. In the same manner, we make $CB_k[i]$ for $k = 2,3,\cdots,\log|\Sigma|$ from the k-th significant bit of $Prev[i]$. Arrays CB_k occupy only $n\log|\Sigma|$ bits. $ST^1_{\sigma i}$'s, $ST^2_{\sigma i}$'s, and LT, which take $O(|\Sigma| \cdot n/\log n)$, $O(|\Sigma| \cdot n\log\log n/\log n)$, and $O(\sqrt{n}\log\log n)$ bits respectively, are the same as our first $O(|P|)$-time algorithm. Hence, the total space becomes $O(n\log|\Sigma| + |\Sigma| \cdot n\log\log n/\log n)$ bits.

When we use arrays B_{σ_i}, we just picked up one word w from B_{σ_i} to count the number of '1's in the last subrange of length less than l. Now we make the same w from arrays CB_k as follows. We pick up one word w_k from each CB_k, $k = 1,2,\cdots,\log|\Sigma|$, and perform bitwise operations on the $\log|\Sigma|$ words. The operations differ from symbol to symbol. For example, if a symbol σ_i is encoded as 10010, then $w = w_1 \& \overline{w_2} \& \overline{w_3} \& w_4 \& \overline{w_5}$, where & is a bitwise AND operation and $\overline{\cdot}$ is a bitwise NOT operation. In this method we need to do bitwise operations $O(\log|\Sigma|)$ times. Thus the time complexity of searching a pattern P becomes $O(|P| \cdot \log|\Sigma|)$. See Figure 7.

```
Algorithm COUNT(σ, i)
  ▷ w, w_i, mask : binary of length l
  ▷ half_l, half_u : binary of length l/2
  N_1 = ST^1_σ[⌊i/L⌋];
  N_2 = ST^2_σ[⌊i/l⌋];
  s = ⌊i/l⌋ · l + 1;  e = ⌊i/l⌋ · l + l;
  w = 2^l − 1;
  for i = 1 to log|Σ|
    w_i = CB_i[s..e];
    if i-th significant bit of σ = '1'
      w = w & w_i;
    else
      w = w & ~w_i;
  mask = 2^l − 2^{l−(i−⌊i/l⌋·l)};
  w = w & mask;
  half_l = w[1..l/2];  half_u = w[l/2 + 1..l];
  N_3 = LT[half_l] + LT[half_u];
  return N_1 + N_2 + N_3;
```

Figure 7 *Count*(σ, i) for $O(|P| \cdot \log|\Sigma|)$-time algorithm

6. Experimental results

For our experiments, we used random texts and biological sequences. We made biological sequences of length 1 and 10 megabytes by cutting or concatenating real DNA($|\Sigma| = 4$) and protein ($|\Sigma| = 20$) sequences, and we generated random texts of length 1 and 10 megabytes in each case when $|\Sigma| = 4$ or 20. Then we also randomly selected patterns from a text, in which the patterns are searched for. We made three groups of 1,000,000 patterns of length between 10 and 20, between 20 and 30, and between 30 and 40. We reversed a half of the patterns to simulate unsuccessful search. We used a machine equipped with dual 2.4GHz Pentium 4 Xeon Processor and 2GByte main memory, running Linux (Red Hat 8.0).

To make a comparison, we implemented two other algorithms due to Abouelhoda et al. (**aok**) [14] and Sim et al. (**skpp**) [15]. Figure 8 shows the space and the running time of each algorithm. Our $O(|P|)$-time algorithm (**ours1**) is the fastest in most cases and our $O(|P| \cdot \log|\Sigma|)$-time algorithm (**ours2**) is the most space-efficient for both $|\Sigma| = 4$ and 20. When $|\Sigma| = 4$, both of our algorithms are faster and more space-efficient than the other two algorithms. When $|\Sigma| = 20$, **ours1** is still more efficient than **skpp** in both time and space. It uses more space than **aok** but it is much faster. When $|\Sigma| = 20$, **ours2** is more efficient than **aok** in both time and space. It still uses less space than **skpp** being comparable in time to **skpp**.

7. Conclusion

We proposed an $O(|P|)$-time algorithm for searching P in a string T of length n over an alphabet Σ. This algorithm uses $O(n \cdot |\Sigma|)$-bits space but it is reasonably small in case of small $|\Sigma|$. We also proposed an $O(|P| \cdot \log|\Sigma|)$-time algorithm using $O(n\log|\Sigma| + |\Sigma| \cdot n\log\log n/\log n)$-bits

	random text						biological sequence					
text length (n)	1M			10M			1M			10M		
pattern length	10-20	20-30	30-40	10-20	20-30	30-40	10-20	20-30	30-40	10-20	20-30	30-40
$\|\Sigma\| = 4$												
algorithm	space	time										
aok	2n	4.45	4.48	4.60	7.08	7.18	7.23	4.38	4.48	4.50	6.95	7.24	7.35
skpp	6n	5.65	5.77	5.81	7.83	8.31	8.25	5.05	5.37	5.55	7.78	8.43	8.74
ours$_1$	0.69n	2.40	2.51	2.62	5.54	5.84	5.92	2.35	2.36	2.40	5.03	5.71	5.91
ours$_2$	0.44n	2.24	2.64	2.39	6.34	6.78	6.90	2.25	2.32	2.53	5.90	6.59	6.88
$\|\Sigma\| = 20$													
algorithm	space	time											
aok	2n	7.71	7.85	7.84	13.69	13.88	13.99	7.39	7.86	7.75	12.91	13.03	13.08
skpp	6n	3.11	3.19	3.22	4.83	4.69	5.08	5.40	7.65	9.12	7.11	9.68	12.89
ours$_1$	3.44n	2.10	2.16	2.18	3.38	3.41	3.37	2.80	3.57	4.45	4.13	5.38	6.54
ours$_2$	1.56n	3.01	3.20	3.16	5.47	5.60	5.56	4.20	5.45	6.50	6.98	9.29	11.33

Figure 8 Comparison of space and running time. We calculated the spaces in bytes through the analysis on data structures in each algorithm and measured the running times of searching for one million patterns in seconds. (**aok**: Abouelhoda et al.'s algorithm, **skpp**: Sim et al.'s algorithm, **ours1** : our $O(|P|)$-time algorithm, **ours2** : our $O(|P| \cdot \log|\Sigma|)$-time algorithm)

space. This is more space-efficient and still fast algorithm. Through experiments we showed that our algorithms are time and space efficient ones in practice.

References

[1] E. McCreight, A space-economical suffix tree construction algorithm, Journal of the ACM, 23(2), pages 262-272, 1976.

[2] P. Weiner, Linear pattern matching algorithms, In Proc. of 14th IEEE Symposium on Switching and Automata Theory, pages 1-11, 1973.

[3] U. Manber and G. Myers, Suffix arrays: A new method for on-line string searches, SIAM Journal on Computing, 22, pages 935-948, 1993.

[4] G. Gonnet, R. Baeza-Yates, and T. Snider, Information Retrieval: Data Structures & Algorithms, Prentice Hall, pages 66-82, 1992.

[5] M. Farach-Colton, P. Ferragina and S. Muthukrishnan, On the sorting-complexity of suffix tree construction, Journal of the ACM, 47(6), pages 987-1011, 2000.

[6] R. Grossi and J.S. Vitter, Compressed suffix arrays and suffix trees with applications to text indexing and string matching, In Proc. of the 32nd ACM Symposium on the Theory of Computing, pages 397-406, 2000.

[7] W. Hon, K. Sadakane, and W. Sung, Breaking a time-and-space barrier in constructing full-text indices, In Proc. of the 44st IEEE Symposium on Foundations of Computer Science, pages 251-260, 2003.

[8] P. Ko and S. Aluru, Space efficient linear time construction of suffix arrays, In Proc. 14th Annual Symposium on Combinatorial Pattern Matching, LNCS 2676, pages 200-210, 2003.

[9] J. Kärkkäinen and P. Sanders, Simple linear work suffix array construction, In Proc. 30th International Colloquium on Automata, Languages and Programming, LNCS 2719, pages 943-955, 2003.

[10] D.K. Kim, J.S. Sim, H. Park, and K. Park, Linear-time construction of suffix arrays, In Proc. 14th Annual Symposium on Combinatorial Pattern Matching, LNCS 2676, pages 186-199, 2003.

[11] K. Sadakane, Succint representation of lcp information and improvement in the compressed suffix arrays, ACM-SIAM Symposium on Discrete Algorithms, pages 225-232, 2002.

[12] E. Ukkonen, On-line construction of suffix trees, Algorithmica, 14(3), pages 249-260, 1995.

[13] P. Ferragina and G. Manzini, Opportunistic data structures with applications, In Proc. of the 41st IEEE Symposium on Foundations of Computer Science, pages 390-398, 2000.

[14] M. Abouelhoda, E. Ohlebusch, and S. Kurtz, Optimal exact string matching based on suffix arrays, In Proc. of the 9th International Symposium on String Processing and Information Retrieval, LNCS 2476, pages 31-43, 2002.

[15] J.S. Sim, D.K. Kim, H. Park and K. Park, Linear-time search in suffix arrays, In Proc. of the 14th Australasian Workshop on Combinatorial

Algorithms, pages 139-146, 2003.
[16] M. Burrows and D. Wheeler, A block sorting lossless data compression algorithm, Technical Report 124, Digital Equipment Corporation, 1994.
[17] I. Munro, Tables, In Proc. of the 16th Conference on Foundation of Software Technology and Theoretical Computer Science, pages 37-42, 1996.

최 용 욱

2002년 서울대학교 전기공학부 학사. 2004년 서울대학교 전기컴퓨터공학부 석사 2004년~현재 Purdue University Dept. of Computer Sciences 박사과정. 관심분야는 컴퓨터 이론 및 알고리즘, 생물정보학

심 정 섭

1995년 서울대학교 컴퓨터공학과 학사 1997년 서울대학교 컴퓨터공학과 석사 2002년 서울대학교 컴퓨터공학부 박사 2002년 3월~2004년 8월 한국전자통신연구원 바이오정보연구팀 선임연구원. 2004년 9월~현재 인하대학교 컴퓨터공학부 전임강사. 관심분야는 컴퓨터 이론 및 알고리즘, 생물정보학

박 근 수

1983년 서울대학교 컴퓨터공학과 학사 1985년 서울대학교 컴퓨터공학과 석사 1991년 미국 Columbia 대학교 전산학 박사. 1991년 11월~1993년 8월 영국 런던대학교 King's College 조교수. 1995년 7월~1995년 8월 호주 Curtin 대학교 방문연구원. 1993년 8월~현재 서울대학교 컴퓨터공학부 교수. 관심분야는 컴퓨터이론, 생물정보학, 암호학