실용적인 접미사 정렬 알고리즘의 개선
(Improvement of Practical Suffix Sorting Algorithm)

정 태 영 † 이 태 형 † 박 근 수 ††
(Taeyoung Jeong) (Taehyung Lee) (Kunsoo Park)

요 약 접미사 배열은 주어진 문자열 내의 모든 접미사를 사전식 순서로 저장하는 자료 구조로, 많은 저장 공간을 사용하는 접미사 트리를 대체하면서 여러 가지 문자열 관련 문제에 사용되고 있다. 이를 $O(n)$ 시간 내에 생성하는 것과 더불어, 실세계 입력에 대하여 작은 시간과 공간을 사용하여 구성하는 알고리즘들 역시 제안되어 왔다. 본 논문은 Maniscalco와 Puglisi[1]가 제안한 접미사 정렬 알고리즘을 분석하고, 프로그램의 수행 시간을 개선한 새로운 알고리즘을 제안한다.

키워드 : 접미사 배열, 접미사 정렬, 다중 키 퀵 정렬

Abstract The suffix array is a data structure storing all suffixes of a string in lexicographical order. It is widely used in string problems instead of the suffix tree, which uses a large amount of memory space. Many researches have shown that not only the suffix array can be built in $O(n)$, but also it can be constructed with a small time and space usage for real-world inputs. In this paper, we analyze a practical suffix sorting algorithm due to Maniscalco and Puglisi [1], and we propose an efficient algorithm which improves Maniscalco-Puglisi's running time.

Key words : Suffix Array, Suffix Sorting, Multi-key Quick Sort

1. 서 론

접미사 배열은 Manber와 Myers[2]에 의해 접미사 트리의 저장 공간을 절약하기 위한 자료 구조로 제시된 이후, 전문 색인(full-text indexing), 압축 등 관련된 여러 문제들에서 기본 자료구조로 광범위하게 사용되고 있다.

이론적으로 접미사 배열은 선형시간 내에 생성될 수 있으나[3-5], 실세계 입력에 대해 이보다 더 적은 수행 시간 및 메모리 공간을 사용하면서 접미사 배열을 생성하는 여러 가지 비선형 알고리즘들이 개발되어 왔다[6]. 예를 들어, Manzini와 Ferragina[7], Maniscalco와 Puglisi[1]가 제안한 알고리즘들은 이론적으로 $O(n^2 \log n)$의 시간 복잡도를 가짐에도 불구하고, 실 세계 입력에 대해 $5 \sim 6n$ byte 정도의 적은 저장 공간과 대단히 빠른 접미사 정렬 속도를 보여주는 것으로 알려져 있다.

본 논문은 Maniscalco와 Puglisi의 알고리즘을 분석하고, 프로그램의 수행 시간을 개선한 새로운 알고리즘을 제안한다. 또한, 이러한 개선이 접미사 정렬의 효율에 어떻게 영향을 주는지를 논하고자 한다.

이 논문은 총 5개의 절로 구성되어 있으며, 2절에서는 배경지식으로서 기존의 실용적인 정렬이 어떠한 방식으로 진행되는지를 설명하고, 3절에서는 이에 대한 개선 방법을 제시할 것이다. 4절에서 이를 실험적으로 분석하고, 마지막으로 5절에서 결론을 내리고자 한다.

2. 배경 지식

2.1 표 기

알파벳 $\Sigma = \{\sigma_0, \sigma_1, ..., \sigma_{|\Sigma|-1}, \}$ 상에서 정의된 길이 n의 문자열 $T = t_0t_1t_2 ... t_{n-1}$ 가 입력으로 주어졌다고 하자. 이

· 본 연구는 기초과학기술연구회(Korea Research Council of Fundamental Science and Technology)의 연구비 지원으로 수행하였음. 이 연구를 위해 서울대학교 컴퓨터 연구소에서 연구장비를 지원하고 공간을 제공함
· 이 논문은 제35회 추계학술대회에서 'Practical Suffix Sorting 알고리즘의 개선'의 제목으로 발표된 논문을 확장한 것임

† 비 회 원 : 서울대학교 전기컴퓨터공학부
tyjeong@theory.snu.ac.kr
thlee@theory.snu.ac.kr
†† 종신회원 : 서울대학교 전기컴퓨터공학부 교수
kpark@theory.snu.ac.kr

논문접수 : 2008년 12월 19일
심사완료 : 2009년 1월 21일

때 i 번째 접미사 $T_i = t_i t_{i+1} \ldots t_{n-1}$, i 번째 문자 $T[i] = t_i$로 정의한다. 문자열 T의 끝을 표시하기 위해 $\$ \notin \Sigma$이고, $\$ < \sigma (\forall \sigma \in \Sigma)$인 문자 $를 사용한다.

두 개의 문자열 α,β에 대하여, 이들의 사전식 대소 관계를 α < β와 같이 표기하기로 한다. 이러한 대소 관계에 기반하여 n개의 접미사 $T_0, T_1, T_2, \ldots, T_{n-1}$을 정렬하여 $T_{SA[0]} < T_{SA[1]} < T_{SA[2]} < \ldots < T_{SA[n-1]}$이 되도록 하는 접미사 배열 SA[0..$n$]을 얻을 수 있다. 임의의 SA[0..$n$]에 대하여, SA[$k$] = i 일 때 ISA[i] = k인 역 접미사 배열을 ISA[0..n]으로 표기하기로 한다.

2.2 접미사 샘플의 선택

실제적으로 빠른 접미사 정렬 알고리즘 중 상당수가 유도 복사(induced copying) 기법을 사용하고 있다[1,6,7]. 유도 복사 기법은 주어진 문자열의 접미사 중 일부만을 문자열 정렬 알고리즘을 이용하여 직접 정렬(direct sort)하고, 그 결과를 이용하여 나머지 접미사들의 사전식 순서를 유도하는 방식이다. 이 때 직접 정렬의 대상이 되는 접미사들을 선택하는 작업을 접미사 샘플링이라 한다.

접미사 샘플링 아이디어를 처음 사용한 Itoh와 Tanaka는 입력 문자열의 접미사들을 아래와 같이 두 개의 집합(또는 샘플)으로 분류한다[8].

$U_{IT} = \{T_i : T[i] < T[i+1]\}$, $V_{IT} = \{T_i : T[i] \geq T[i+1]\}$

Ko와 Aluru는 이를 개선하여, 전체 접미사들을 다음과 같이 분류하였다[5].

$U_{KA} = \{T_i : T_i < T_{i+1}\}$, $V_{KA} = \{T_i : T_i > T_{i+1}\}$

이렇게 분류된 두 개의 접미사 집합 중 크기가 작은 집합을 직접 정렬하여 해당 집합 내 접미사들의 순서를 결정하면, 이를 이용하여 전체 접미사 배열을 유도할 수 있다[5,7-9]. 이 때, 전체 수행 시간 중 직접 정렬에 소요되는 시간의 비중이 크기 때문에 선택되는 접미사 샘플의 크기가 작을수록 전체 알고리즘의 수행 시간도 짧아지게 된다.

Mori[9]는 Ko와 Aluru의 분류 조건을 변형하여, 더 작은 크기의 접미사 집합을 샘플링하는 방법을 제시하였다. Mori의 접미사 분류 조건은 다음과 같다.

$S_M = \{T_i : T_i \in U_{KA},\ T_{i+1} \in V_{KA}\}$

문자열 T = edabdccdeedab를 이들 세 가지 기준으로 분류한 결과는 표 1과 같다. ★가 표시된 칸은 해당하는 접미사가 S_M에 속함을 의미한다.

표 1 접미사 샘플링 결과

T	e	d	a	b	d	c	c	d	e	e	d	a	b	$\$$
IT	V	V	U	U	V	V	U	U	V	V	V	U	V	-
KA	V	V	U	U	V	U	U	U	V	V	V	U	V	-
M				★				★				★		

2.3 직접 정렬

Maniscalco와 Puglisi의 알고리즘(이하 MP 알고리즘)은 위와 같이 뽑힌 S_M 접미사들(최대 $n/2$개)을 버킷 정렬(Bucket Sort)와 다중 키 퀵 정렬 (Multi-Key Quick Sort, 이하 MKQS)[10,11]를 사용하여 직접 정렬한다. 각 문자열의 첫 k글자가 동일한(Least Common Prefix(LCP)의 길이가 k인) 문자열의 집합 A가 주어졌을 때, MKQS는 이 집합 내에서 임의의 문자열 p∈A를 선택하고 p의 k+1 번째 문자 p[k]를 분할 원소(pivot)로 하여 A를 세 개의 부분 집합 $A_<$, $A_=$, $A_>$로 분할한다. (y∈$A_<$이면 y[k] < p[k], y∈$A_=$이면 y[k] = p[k], y∈$A_>$ 이면 y[k] > p[k]) 이 세 집합은 다시 MKQS을 재귀적으로 호출하여 정렬된다.

MKQS의 pseudo code는 다음과 같다.

```
1  MKQS(A, k)
2     if |A| < THRESHOLD
3        InsertionSort(A, k)
4     else
5        choose a pivot suffix p
6        partition A into A<, A=, A> with p[k]
7        MKQS(A<, k)
8        MKQS(A=, k+1)
9        MKQS(A>, k)
10    fi
```

2.4 ISA'의 사용

MP 알고리즘은 또한 이러한 비교 작업의 효율을 높이기 위하여 입력으로 주어진 문자열 T를 정수 배열 ISA'으로 변환하여 비교에 이용한다. 이러한 변환 작업은 각각의 bigram T[i..i+1]을 2 byte 정수로 취급하고 아래와 같은 과정을 거쳐 생성된 code(i)와 대응시키는 것으로 이루어진다.

- 각 bigram이 T 내에서 발견되는 횟수를 bi_freq[0..2^{16}-1] 내에 저장한다.
- lo_rank[0..2^{16}-1]와 hi_rank[0..2^{16}-1]를 다음과 같이 계산한다.
  - lo_rank[0] = 0
  - 각각의 bigram b에 대하여,
    hi_rank[b] = lo_rank[b] + bi_freq[b] + k. k는 hi_rank[b] mod 256=0을 만족하는 최소의 정수.
    lo_rank[b + 1] = hi_rank[b]
- code(i) = hi_rank[T[i..i+1]] + T[i+2]

이와 같은 대응 관계를 통하여 ISA'은 각 접미사의 처음 세 글자에 대한 사전식 순서 정보를 보유하면서 동시에 T의 모든 정보를 보존할 수 있다. 또한 정렬이

완료된 접미사에 대하여 ISA′의 값을 유일한 값으로 수정함으로써, 이후 접미사의 대소 관계를 비교할 때 이미 정렬된 접미사에 대한 비교의 양을 크게 줄일 수 있다.

2.5 반복 구간에 대한 처리

문자열 T에서 문자열 u가 r번 연속적으로 나타날 때, 즉, T의 부분 문자열 $T[i..j] = T[i]T[i+1]...T[j]$가 u^r과 같은 형태로 나타나는 경우, 두 개의 문자열 u^r과 u^{r-1}의 대소관계를 결정하기 위해서는 최소한 $|u^{r-1}| = |u|$ $(r-1)$개의 문자를 비교할 필요가 있다. MP 알고리즘은 이러한 불필요한 연산을 줄이기 위하여 다음과 같은 휴리스틱을 사용한다.

MKQS가 LCP 길이 k까지 진행되었을 때, 가운데 부분 집합 V에 접미사 T_i와 접미사 T_{i+k}가 동시에 존재한다면, T_i는 반복된 문자열 접두사(prefix)를 갖는다.

Observation 1. $T_i \in V$, $T_{i+k} \in V$라면, T_i는 $T[i...i+k-1]$가 반복된 문자열을 접두사로 갖는다.

그러나 이를 MKQS의 수행 중에 확인하기 위해서는 $O(|V|^2)$의 수행시간이 필요하다. 때문에 MP 알고리즘은 다음과 같은 약화된 조건을 이용한다.

Observation 2. $T[i] = T[i+k]$라면, T_i는 $T[i...i+k-1]$가 반복된 문자열을 접두사로 가질 가능성이 있다.

이를 통해 가능성 있는 반복이 발견되었을 때 이를 확인하고 처리하는 과정은 다음과 같다.

- V에 포함된 모든 접미사 T_i에 대하여 $ISA'[i] = \xi$로 설정한다.
- V를 ξ를 분할 원소로 하여 $V_<$, $V_=$, $V_>$으로 분할한다. $V_=$에 포함되는 접미사들이 실제 반복을 일으키는 접미사들이다.
- $V_<$과 $V_>$를 MKQS를 이용하여 재귀적으로 정렬한다.
- $V_<$과 $V_>$의 정보를 이용하여 $V_=$을 구성한다.

3. 개선된 알고리즘

3.1 변경점

MP 알고리즘은 기존의 직접 정렬에 기반한 다른 접미사 배열 생성 알고리즘들에 비해 빠른 속도와 적은 메모리 사용량을 보여준다. 그러나 MP 알고리즘은 $6n$ byte의 메모리 사용량을 확보하기 위하여 입력으로 주어진 문자열 T의 저장 공간을 덮어쓰는 형식을 취하며, 이 정보를 보존하기 위하여 ISA′ 내부에 문자열의 정보를 사상하고, MKQS에 의한 정렬이 끝난 후 이를 다시 복구하는 등의 작업을 추가로 요구한다.

우리는 이러한 과정을 단순화하고 알고리즘의 수행시간을 개선하기 위하여, n byte의 추가 공간을 사용하여 입력 문자열을 보존하는 대신, ISA′의 사상을 단순화하는 방법을 제안한다. 2-4절에서 언급한 code(i)의 생성 알고리즘을 참고하면 이는 다음과 같이 표기할 수 있다.

- 기존 : code(i) = hi_rank[$T[i..i+1]$]+$T[i+2]$
- 변경 : code(i) = hi_rank[$T[i..i+1]$]

3.2 변경에 의한 효과

이러한 변경에 의해 직접적으로 얻을 수 있는 이득은, 알고리즘 여러 곳에서 쓰이고 있는 문자열 정보 관련 처리 내용을 일원화할 수 있다는 점을 들 수 있다. MP 알고리즘은 ISA′ 내에 원본 문자열 T에 관한 정보를 보존하기 위하여 최초에 ISA′ 내에 정보를 저장하고, 수행 중 유일한 ISA′ 값에 의해 정보가 손상되는 것을 막기 위해 자료를 이동시키며, MKQS가 끝난 이후 이를 다시 원본 문자열로 복구하는 과정을 거친다. 이러한 과정들이 제거되는 것은 3~5%정도의 성능 향상을 가져온다.

또한 이러한 변경은 MKQS의 진행에도 영향을 준다. MP 알고리즘 내에서 이용되는 MKQS는 각 접미사의 첫 bigram을 비교의 대상으로 하며, 가운데 부분 집합에 대해서도 bigram의 크기인 2만큼씩 LCP 길이를 증가시켜 가면서 비교하는 것을 원칙으로 하고 있다. 그러나 접미사의 세 번째 글자에 대한 정보까지 갖고 있는 ISA′이 비교의 대상으로 이용되면서 사실상 MKQS가 trigram에 의해 분할을 수행하는 형태가 되며, ISA′을 이용하지 않고 단순히 bigram을 비교했다면 같은 부분 집합에 속했을 접미사들이 실제로 MKQS의 분할 이후 같은 가운데 부분 집합에 속하지 못하게 되는 결과를 낳고 있다.

예를 들어, MKQS를 진행하는 과정에서 다음과 같은 접미사 집합을 얻었다고 가정해보자.

S = {aaa..,aab..,aac..,aad..,aae..,aba..,abb..,abc..,
abd..,abe..,aca..,acb..,acc..,acd..,ace..}

중간값인 p=ab를 분할 원소로 이용하여 S를 정렬하면, S는 다음과 같은 세 개의 부분 집합으로 분할된다. 이 LCP에서 필요한 MKQS를 완료하는 데에는, 첫 분할과 함께 각 집합이 정렬되었는지를 확인하는 두 번째 분할까지 총 15+5+5=25번의 원소 접근이 필요하다.

$S_<$ = {aaa..,aab..,aac..,aad..,aae..}
$S_=$ = {aba..,abb..,abc..,abd..,abe..}
$S_>$ = {aca..,acb..,acc..,acd..,ace..}

그러나 동일한 S에 ISA′을 사용하여 정렬을 시도하면, 중간 값 p=abc를 분할 원소로 선택하여 첫 분할이 다음과 같은 형태로 진행된다.

$S'_<$ = {aaa..,aab..,aac..,aad..,aae..,aba..,abb..}
$S'_=$ = {abc..}
$S'_>$ = {abd..,abe..,aca..,acb..,acc..,acd..,ace..}

또한, 분할이 완료되지 않은 $S'_<$과 $S'_>$에 대하여, 각각 3번씩의 분할이 추가로 필요하며, 최종적으로 총 7번

의 분할과 $15+7\times2+3\times4+1\times8=49$번의 원소 접근을 요구하여, ISA′을 사용하지 않았을 때에 비해 더 많은 시간과 자원을 소비하게 된다.

일반적인 접미사 정렬의 경우 이 예제보다 bigram의 숫자가 많으며, $S_=$에 비하여 $S_<$과 $S_>$의 크기가 상대적으로 크다는 점을 감안한다면, ISA′에 접미사의 처음 trigram에 대한 정보를 갖고 MKQS를 진행하는 경우, ISA′이 한 bigram에 대한 정보를 갖고 있을 때보다 동일한 진행 상태에 도달하기 위한 분할의 횟수나 접근하는 원소의 숫자가 많아진다고 추측할 수 있다.

4. 실험 결과

4.1 실험 환경

- Intel Pentium 4 2.60GHz CPU. 2GB RAM
- Windows XP SP2
- Microsoft Visual Studio 2005

4.2 Corpus 문서에 대한 실행 결과

표 2와 같은 4개 문서에 대하여, 개선된 알고리즘(ours)과 MP 알고리즘(mp), 그리고 Mori[9]의 알고리즘(mori)을 적용하여 실행시킨 결과는 그림 1과 같다. 개선된 알고리즘은 기존의 MP 알고리즘에 비해서 약 10% 이상의 속도 향상을 보이고 있는 것을 확인할 수 있었으며, 현재까지 가장 빠른 속도를 보이는 것으로 알려진 Mori의 알고리즘과도 비슷한 수행 속도를 보이는 것이 관측되었다. 또한 MP 알고리즘과의 MKQS 수행 속도 비교에서도 그림 2와 같이 10~12% 정도의 향상을 확인할 수 있었다.

표 2 실험에 사용된 Corpus 문서

	Size (byte)	$\|\Sigma\|$	내용
howto	39,422,105	197	Linux Howto Files
dna	34,553,758	4	Human chromosome 22
rfc	116,421,901	120	IETF RFC files
w3c2	104,201,579	255	W3C site HTML files

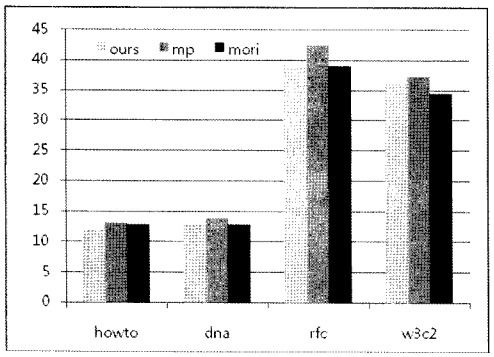

그림 1 실행 속도 비교(단위: 초)

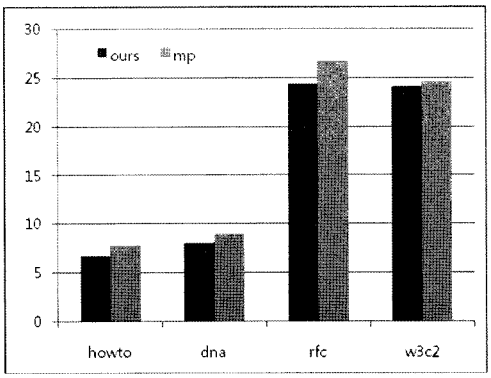

그림 2 MKQS의 실행 속도 비교(단위: 초)

4.3 howto 문서에 대한 세부 실행 결과 분석

ISA′의 개선에 따른 분할의 변화를 관측하기 위하여, MKQS가 각 깊이(depth)에서 분할을 위하여 사용하는 ISA′의 메모리 접근 횟수를 측정해 보았다. 그림 3은 개선 전후의 알고리즘이 각 MKQS 깊이에 도달하기까지 ISA′의 메모리 접근 횟수를 나타낸 것이다. 이 수치는 각 깊이 별 접근 회수의 누적도수 형태를 보여주고 있으며, ISA′이 개선된 후 임의의 깊이에 도달하기까지 분할에 사용되는 비용이 현저하게 감소하였음을 확인할 수 있게 해 준다.

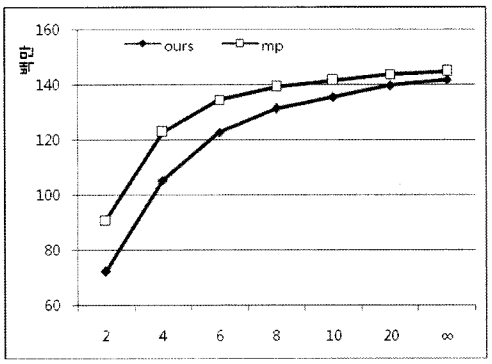

그림 3 각 깊이에 도달하기까지 ISA′의 메모리 접근 횟수

5. 결 론

우리는 MP 알고리즘이 사용하는 정보 저장 방식을 수정하여 MP 알고리즘이 갖고 있는 메모리 사용 방식과 MKQS의 진행 양상을 개선시켰다. 이러한 수정은 MKQS의 진행 상에서 사용되는 자원의 양을 절감할 수 있으며, 적지 않은 성능 향상을 가져오는 것이 실험적으로 확인되었다. 또한 기수 정렬(Radix Sort) 등의 다른 정렬 방법과의 연동을 고려하거나, MKQS가 종료된 후, S_M으로 샘플링 되지 않은 접미사들을 정렬하는 방법을

최적화하는 등의 개선의 여지가 많이 남아 있어, 앞으로의 추가적인 연구 과제가 될 것이라 본다.

참 고 문 헌

[1] Maniscalco, M., and Puglisi, S. Faster lightweight suffix array construction. In *Proceedings of 17th Australasian Workshop on Combinatorial Algorithms*, pp. 16-29. 2006.

[2] Manber, U., and Myers, G. Suffix arrays: a new method for on-line string searches. *SIAM Journal of Computing 22*, 5, pp. 935-948. 1993.

[3] Kärkkäinen, J., Sanders, P., and Burkhardt, S. Linear work suffix array construction. *Journal of the ACM 53*, 6, pp. 918-936. 2006.

[4] Kim, D. K., Sim, J. S., Park, H., and Park, K. Constructing suffix arrays in linear time. *Journal of Discrete Algorithms 3*, 2-4, pp. 126-142. 2005.

[5] Ko, P., and Aluru, S. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms 3*, 2-4, pp. 143-156. 2005.

[6] Puglisi, S. J., Smyth, W. F., and Turpin, A. H. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys 39*, 2, 2007.

[7] Manzini, G., and Ferragina, P. Engineering a lightweight suffix array construction algorithm. *Algorithmica 40*, 1, pp. 33-50. 2004.

[8] Itoh, H., and Tanaka, H. An efficient method for in memory construction of suffix arrays. In *Proceedings of the String Processing and Information Retrieval*, p. 81. 1999.

[9] Mori, Y. Divsufsort (version 1.2.3). http://www.homepage3.nifty.com/wpage/software/livbdivsufsort.html, 2005.

[10] Bentley, J. L., and McIlroy, M. D. Engineering a sort function. *Software-Practice & Experience 23*, 11, pp. 1249-1265. 1993.

[11] Bentley, J. L., and Sedgewick, R. Fast algorithms for sorting and searching strings. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*. pp. 360-369, 1997.

정 태 영

2007년 서울대학교 컴퓨터공학부 학사 2007년~현재 서울대학교 전기컴퓨터공학부 석사과정. 관심분야는 컴퓨터 이론, 알고리즘

이 태 형

2003년 서울대학교 컴퓨터공학부 학사 2003년~현재 서울대학교 전기컴퓨터공학부 박사과정. 관심분야는 컴퓨터 이론, 알고리즘

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