압축된 써픽스 배열 구축의 실제적인 성능 비교
(Comparisons of Practical Performance for Constructing Compressed Suffix Arrays)

박 치 성 [†] 김 민 환 [††] 이 석 환 [†††] 권 기 룡 [††††] 김 동 규 [†††††]
(Chi Seong Park) (Min Hwan Kim) (Suk Hwan Lee) (Ki Ryong Kwon) (Dong Kyue Kim)

요 약 써픽스 배열은 기본적인 전체 텍스트 인덱스 자료구조로서, 반복되는 패턴 질의 수행 시 효율적으로 사용될 수 있다. 유용한 전체 텍스트 인덱스 자료구조들이 많이 제안되어왔음에도 불구하고, $O(n\log n)$-비트 공간을 필요로 하는 공통적인 문제점으로 인하여 보다 효율적으로 공간을 사용할 수 있는 방법에 대한 필요성이 요구되었다. 하지만 기 개발된 압축된 써픽스 배열이나 FM-인덱스와 같은 것들 또한 이미 존재하는 써픽스 배열에서부터 구축되어야 하기 때문에 실제적인 사용 공간을 줄일 수는 없었다. 최근, 써픽스 배열을 구축할 필요 없이 텍스트로부터 직접 압축된 써픽스 배열을 구축할 수 있는 두 가지 알고리즘들이 제안되었다. 본 논문에서는 실험을 통해 자료구조 구축 시간과 구축 시 필요로 하는 최대 사용 공간, 구축이 끝난 후 최종 자료구조의 크기 등을 측정함으로써 이 두 가지 압축된 써픽스 배열 구축 알고리즘과 기존의 써픽스 배열들과의 실제적인 성능을 비교한다.

키워드 : 전체 텍스트 인덱스 자료구조, 써픽스 배열, 압축된 써픽스 배열, 구현 이슈, 실제적인 성능, odd-even 기법, skew 기법

Abstract Suffix arrays, fundamental full-text index data structures, can be efficiently used where patterns are queried many times. Although many useful full-text index data structures have been proposed, their $O(n\log n)$-bit space consumption motivates researchers to develop more space-efficient ones. However, their space efficient versions such as the compressed suffix array and the FM-index have been developed; those can not reduce the practical working space because their constructions are based on the existing suffix array. Recently, two direct construction algorithms of compressed suffix arrays from the text without constructing the suffix array have been proposed. In this paper, we compare practical performance of these algorithms of compressed suffix arrays with that of various algorithms of suffix arrays by measuring the construction times, the peak memory usages during construction and the sizes of their final outputs.

Key words : full-text index data structure, suffix array, compressed suffix array, implementation issue, practical performance, odd-even scheme, skew scheme

· The preliminary version of this paper was presented in WAAC'2006.
· 본 연구는 한국과학재단 특정기초연구(R01-2006-000-10260-0) 지원으로 수행되었음
· 본 연구는 정보통신부 및 정보통신연구진흥원의 대학 IT연구센터 지원사업의 연구결과로 수행되었음(IITA-2006-(C1090-0602-0011))

† 학생회원 : 부산대학교 컴퓨터공학과
cspark@esslab.hanyang.ac.kr
†† 종신회원 : 부산대학교 컴퓨터공학과 교수
mhkim@pusan.ac.kr
††† 정 회 원 : 동명대학교 정보보안학과 교수
skylee@tit.ac.kr
†††† 정 회 원 : 부경대학교 컴퓨터공학과 교수
krkwon@pknu.ac.kr
††††† 종신회원 : 한양대학교 전자통신컴퓨터공학부 교수
dqkim@hanyang.ac.kr
(Corresponding author임)

논문접수 : 2007년 1월 22일
심사완료 : 2007년 3월 15일

1. 서 론

Given a text string T of length n over an alphabet Σ and a pattern string P of length m, the pattern matching problem is finding all instances of P in T. The studies for efficient pattern matching are divided into two approaches: One is to preprocess the P in $O(m)$ time and then search in $O(n)$ time. The other is to build a full-text index data structure for T in $O(n)$ time and then search in $O(m)$ time.

The latter approach is more appropriate than the former when we search DNA sequences in full genome sequences since the text is much longer than the pattern and we have to search many patterns in the text.

Two well-known such index data structures are suffix trees [1-5] and suffix arrays [6-10]. The suffix tree is a compacted trie of all suffixes of the text, and the suffix array is a lexicographically sorted list of all the suffixes of the text. The suffix tree can be constructed in $O(n)$ time due to McCreight [2], Ukkonen [3], Farach [4] and so on. The suffix array can be also constructed in $O(n)$ time due to Kim et al. [8], Ko and Aluru [9], and Kärkkäinen and Sanders [10]. In addition, Kim et al. [11], Larsson and Sadakane [12], Manzini and Ferragina [13], and Schürmann and Stoye [14] also proposed construction algorithms which are not linear time but work practically fast on some environments.

Although many useful full-text index data structures were developed, their space consumption ($O(n\log n)$ bits for a string of length n) which is larger than the text itself motivates researchers to develop more space efficient ones. Munro et al. [15] developed a succinct representation of a suffix tree topology using $O(n\log\Sigma)$-bit working space under the name of space efficient trees. Grossi and Vitter [16] developed the compressed suffix array using $O(n\log\Sigma)$-bit working space. Ferragina and Manzini [17] suggested opportunistic data structures using $O(n\log\Sigma)$-bit working space under the name of FM-index. Nevertheless, they can't reduce the actual working space less than $O(n\log n)$ bits because their algorithms are based on a suffix array as an input.

Recently, two algorithms that construct directly the compressed suffix arrays from given texts in about $O(n\log\Sigma)$-bit space have been developed. Hon et al. [18] and Na [19] construct the compressed suffix arrays without constructing a suffix array that follows the odd-even scheme of Farach et al. [4,5] and Kim et al. [8,11], and the skew scheme of Kärkkäinen and Sanders [10], respectively. In this paper, we compare practical performance of these algorithms of compressed suffix arrays with that of various algorithms of suffix arrays by measuring the construction times, the peak memory usages during construction and the sizes of their final outputs.

We give some previous algorithms of constructing suffix arrays in Section 2. In Section 3, we describe two algorithms that directly construct the compressed suffix arrays from the text and discuss some implementation issues. In Section 4, we show the experimental results and compare the algorithms described in Section 2.

2. Constructing Suffix Arrays

We first introduce several existing algorithms of constructing suffix arrays in this section. Table 1 shows the summary of time and space performances.

• Manber and Myers [6] presented a radix-sorting

Table 1 Previous Construction Algorithms

category	author	name (code name)	year	asymptotic worst-case time	working space[a] (4-byte word integer)
suffix array	Manber and Myers[6]	-	1993	$O(n\log n)$	$8n$ bytes
	Kim et al. [8]	-	2003	$O(n)$	$O(n\log n)$ bits
	Ko and Aluru [9]	-	2003	$O(n)$	$12n$ bytes + $2.5n$ bits
	Kärkkäinen and Sanders [10]	*skew(sa)*	2003	$O(n)$	$O(n\log n)$ bits
	Kim et al. [11]	*odd-even(sa)*	2004	$O(n\log\log n)$	$O(n\log n)$ bits
	Larsson and Sadakane [12]	*qsufsort*	1999	$O(n\log n)$	$8n$ bytes
	Manzini and Ferragina [13]	*deep-shallow*	2004	$O(n^2\log n)$	$5.03n$ bytes
	Schürmann and Stoye [14]	*bpr*	2005	$O(n^2)$	$8n$ bytes
compressed suffix array	Hon et al. [18]	*odd-even(csa)*	2003	$O(n\log\log\|\Sigma\|)$	$O(n\log\|\Sigma\|)$ bits
	Na [19]	*skew(csa)*	2005	$O(n)$	$O\left(n\log\|\Sigma\|\cdot\log^{\varepsilon}_{\|\Sigma\|}n\right)$ bits

a) The values in this column can be varied under different implementation environments.

based algorithm. They used the doubling technique introduced by Karp et al. [20], which reduces the number of sorting passes.

- Kim et al. [8] followed Farach et al. [4,5]'s odd-even scheme, i.e., the recursive divide-and-conquer scheme that divides the suffixes of T into odd suffixes and even suffixes. It first constructs the suffix array SA_o for odd suffixes of T, and constructs the suffix array SA_e for even suffixes of T from SA_o, and then merges SA_o and SA_e to construct the suffix array for all suffixes of T. Ko and Aluru [9] and Kärkkäinen and Sanders [10] were also based on the recursive divide-and-conquer scheme. Although odd-even scheme has some advantages over the Kärkkäinen and Sanders' skew scheme such as less recursive calls and fast encoding, Kim et al.'s algorithm is slow overall because of the complicated merging steps. Kim, Jo and Park [11] adopted a fast odd-even merging algorithm to Kim et al. [8] using the backward search in $O(n\log\log n)$ time. This is a more practical algorithm that constructs the suffix arrays fast when the alphabet is fixed-size.
- Ko and Aluru [9] divided the suffixes of T into S-type and L-type suffixes through a scan of T. This is a linear time version of Itoh and Tanaka [21]'s algorithm.
- Kärkkäinen and Sanders [10] used skew scheme, i.e., divided the suffixes of T into suffixes beginning at positions $i \bmod 3 \neq 0$ and the other suffixes beginning at positions $i \bmod 3 = 0$. The merging step of this scheme is simple and fast.
- Larsson and Sadakane [12] used the doubling technique as in Manber and Myers'. However, this is a more practical algorithm by removing unnecessary scanning and idle reorganizing of already sorted suffixes.
- Manzini and Ferragina [13] combined different methods to sort suffixes depending on LCP (Longest Common Prefix) lengths and spend quite a bit of work on finding suitable settings to achieve fast construction. Although the time complexity is rather poor, it shows best performance in many cases.
- Schürmann and Stoye [14] combined the approach of refining groups with equal prefixes by recursively performing radix steps and the pull technique. This algorithm is not complicated and very fast with growing average LCP.

3. Constructing Compressed Suffix Arrays

This section introduces the odd-even scheme and the skew scheme for compressed suffix arrays, and discusses some implementation issues.

3.1 Succinct representation of Ψ function

We first introduce the Ψ function which is the main component of the compressed suffix array. A text T of length n over alphabet Σ is denoted by $T[0...n-1]$. Let S_i for $o \le i \le n-1$ denote the ith suffix of T. The suffix array $SA_T[0...n-1]$ of T is an array of integers such that $T[SA[i]...n]$ is lexicographically the ith smallest suffix of T. The Ψ_T function is defined as follows:

$$\Psi_T = \begin{cases} SA_T^{-1}[SA_T[i]+1] & \text{if} \quad SA_T[i] \neq n-1 \\ SA_T^{-1} & \text{otherwise} \end{cases}$$

The Ψ_T function can be succinctly represented in $O(n\log\Sigma)$ bits using unary codes so that each Ψ_T can be retrieved in constant time. However, we should call rank and select functions many times which are base operations of succinct representation to access to Ψ_T, and the performance of rank and select functions have much effect on the overall construction time of compressed suffix arrays. We import fast rank and select functions proposed in Kim et al. [22] to minimize the side effects of function calls. Those are byte-based implementations for modern computers of which atomic units are bytes.

3.2 Odd-even scheme

Hon, Sadakane and Sung's algorithm [18] constructed the compressed suffix array directly from T using odd-even scheme as follows. Let $h = \lceil \log_2 \log_{|\Sigma|} n \rceil$. For $0 \le i \le h$, T^i is defined as the string over the alphabet Σ^{2^i}, which is formed by concatenating every 2^i characters in T to make one character. The basic framework of the algorithm is to use a bottom-up approach to construct Ψ_{T^i}, for $i=h$ down to 0, thereby obtaining Ψ of T in the end. Precisely,

```
1. For $i = h$
   Construct suffix array $SA_{T^i}$ for $T^i$
   Construct $\Psi_{T^i}(=\Psi_{T_e^{i-1}})$ from $SA_{T^i}$
2. For $i = h-1$ to 0
   (a) Construct $\Psi_{T_o^i}$ for $\Psi_{T_e^i}$
   (b) Merge $\Psi_{T_o^i}$ and $\Psi_{T_e^i}$ into $\Psi_{T^i}$ using backward search algorithm
```

The compressed suffix array consists of SA_{T^i} and Ψ_{T^i} in every step is constructed when the step 0 is finished. The entire procedure takes $O(n \log\log_{|\Sigma|} n)$ time using $O(n\log\Sigma)$-bit working space.

We encountered two problems during implementing Hon et al.'s algorithm as follows:

- We could not construct the suffix array SA_{T^h} using one of generic algorithms since the size of a character in T^h, i.e., concatenated 2^h characters of T in hth step, may exceed one 4-byte word. Suppose that T of length n=128K=2^{17} over $|\Sigma|$=2 is given. Then $h = \lfloor \log\log_{|\Sigma|} n \rfloor = 5$. The alphabet of T^5 (namely, Σ^{2^5} in the 5th step extends to $2^{2^5} = 2^{32}$. So it is beyond the capability of a 4-byte word, and it is impossible to construct SA_{T^h} under these conditions. Moreover, the number of operations required by stable sorting which is internally used to construct Ψ functions could be large, since we sort all characters in stable sorting and the size of the alphabet is very large. Sometimes, the size of an alphabet can exceed main-memory size. (e.g., 2^{32}) Both of these problems are due to exponentially increased size of concatenated characters.
  To remedy this, we adapt the encoding process for reducing the size of a character in T^h. We use a *long long integer* type (8 bytes) supported in the ISO C99 standard temporarily when the size of a character exceeds 2^{32}.
- Hon et al.'s algorithm uses an $O(n+|\Sigma|)$-bit auxiliary data structure for efficient rank queries to improve backward search algorithm from $O(n\log n)$ time to $O(n\log\log\Sigma)$ time. The backward search algorithm performs rank queries for characters to find patterns within the text. The rank queries in Hon et al.'s are based on specific ranges of the Ψ function. If the length of range is smaller than $\log|\Sigma|$, the required rank can be found in $O(\log\log|\Sigma|)$ time by performing simple binary searching. Otherwise, the auxiliary data structure is needed to support $O(\log\log|\Sigma|)$-time backward search step. However, it is difficult to implement the auxiliary data structure since it supports perfect hashing [23] and log-logarithmic worst-case range queries [24]. We did not use auxiliary data structures but perform binary searching in all cases.

We implemented these alternatives in order not to decrease the inherent performance as carefully as possible.

3.3 Skew scheme

Na [19]'s algorithm used a bottom-up approach similar with Hon et al.'s except that it used skew scheme instead of odd-even scheme. The entire procedure takes $O(n)$ time using $O(n\log|\Sigma|\cdot\log_{|\Sigma|}^{\varepsilon} n)$-bit working space. This algorithm basically uses encoding process and has $\lfloor \log_3 \log_{|\Sigma|} n \rfloor$ steps, which is smaller than Hon et al.'s $\lfloor \log_2 \log_{|\Sigma|} n \rfloor$ steps. We had nothing particular to consider alternatives during implementing this algorithm.

4. Experimental Results

In this section, we show some experimental results comparing five algorithms for constructing suffix arrays (i.e., *skew(sa)*, *odd-even(sa)*, *qsufsort*, *deep-shallow* and *bpr*) with two algorithms for constructing compressed suffix arrays (i.e., *odd-even(csa)* and *skew(csa)*) described in Table 1. *Skew(sa)*1), *qsufsort*2), *deep-shallow*3) and *bpr*4) are provided by each authors, and *odd-even(sa)*, *odd-even(csa)* and *skew(csa)* are implemented by ourselves. We measure the construction times, peak

1) http://www.mpi-inf.mpg.de/~sanders/programs/suffix/
2) http://www.larsson.dogma.net/research.html
3) http://www.mfn.unipmn.it/~manzini/lightweight/
4) http://bibiserv.techfak.uni-bielefeld.de/bpr/

memory usages during construction, and the sizes of final outputs on a machine equipped a 3.0Ghz Intel Pentium IV with 512MB main memory. The machine is operated by Linux (Fedora Core 4) and we use gcc/g++ for compilation. The extra program[5] is used to track the algorithm's memory usages. In all experiments, we used random strings which are varied in lengths 1M, 5M, and 10M, and the size of alphabets 8 and 20.

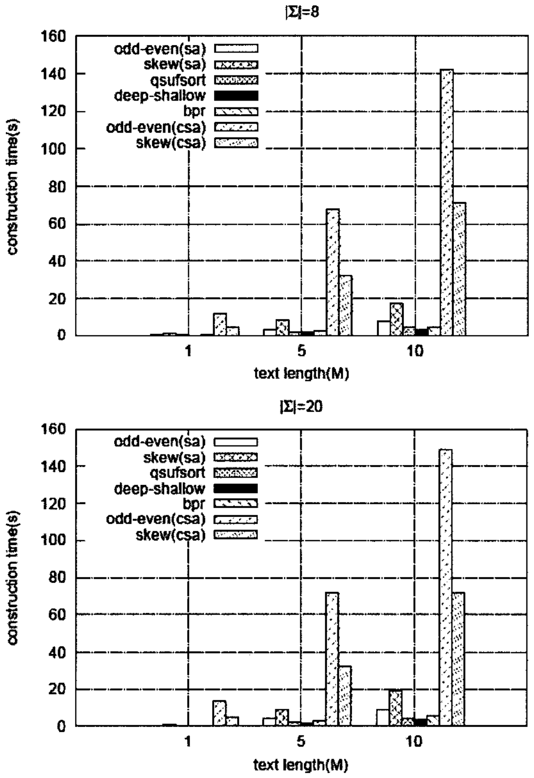


Figure 1 Construction time of suffix arrays and compressed suffix arrays when |Σ|=8 and |Σ|=20.

Figure 1 shows the construction times of algorithms. The constructions of compressed suffix arrays are much slower than the constructions of suffix arrays in all cases. *Odd-even(csa)* is about 2 times slower than *skew(csa)* since the additional encoding steps and stable sorting operations described in Section 3.2 takes much time.

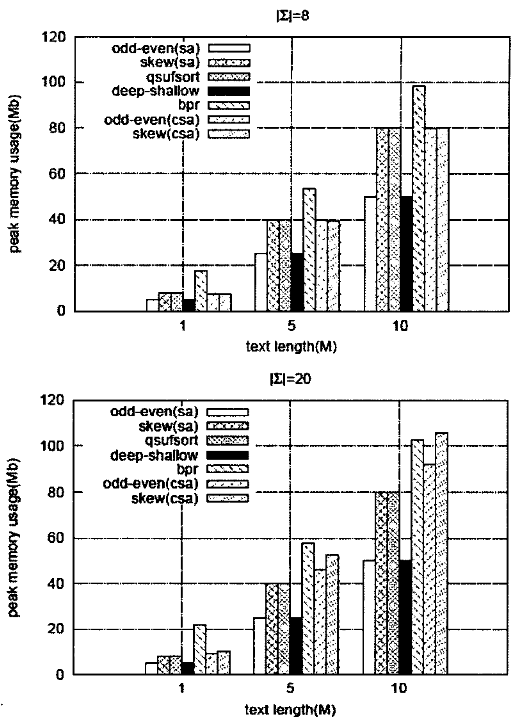


Figure 2 Peak memory usages during constructing suffix arrays and compressed suffix arrays when |Σ|=8 and |Σ|=20.

Figure 2 shows the peak memory usages which are maximum working spaces during constructing each algorithm. *Odd-even(csa)* and *skew(csa)* uses similar working space when |Σ|=8 and *odd-even (csa)* uses a little smaller working space than *skew(csa)* by 11~15% when |Σ|=20. However, both of them require much space than most of suffix arrays in most cases.

Figure 3 shows the sizes of suffix arrays and compressed suffix arrays when all construction steps are finished. We measure only *pos* array in suffix arrays, and SA_{T^k} and Ψ functions of all steps in compressed suffix arrays. The size of *odd-even (csa)*'s is a little larger than others, and the size of *skew(csa)*'s is similar with that of suffix arrays when |Σ|=8. However, the sizes of two compressed suffix arrays are increasingly larger than that of suffix arrays with large alphabets. Although compressed suffix arrays are proposed for space efficiency, construction algorithms of compressed

5) http://prj.softpixel.com/mcd/

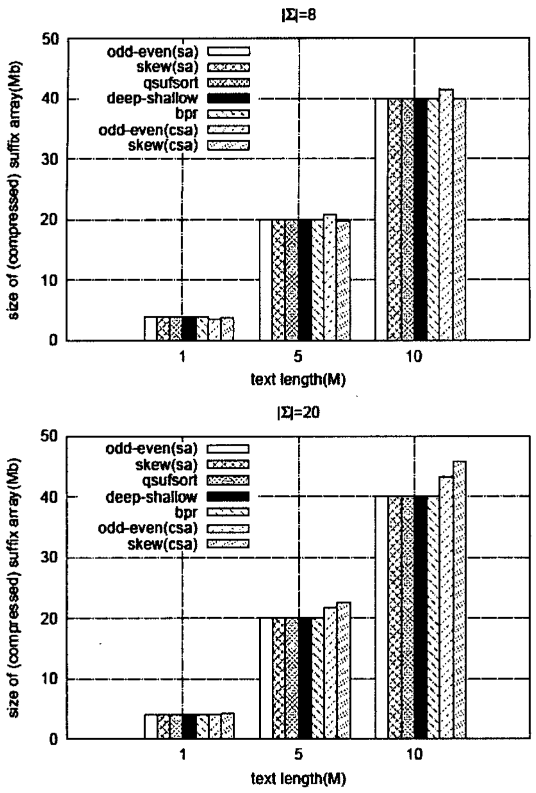


Figure 3 Size of sorted arrays for suffix arrays and compressed suffix arrays when |Σ|=8 and |Σ|=20.

suffix arrays require much space as well as much construction time than those of suffix arrays, using $O(n\log n)$-bit working space.

5. Conclusion

The complexity in time and space of compressed suffix arrays is about $O(n)$ and $O(n\log|\Sigma|)$bits. On the other hand in suffix arrays, time complexity is more than $O(n)$ and $O(n\log n)$ bits are required for space. As it can see from the above, compressed suffix arrays are better than suffix arrays, theoretically. However, we encountered to practical problems while implementing the compressed suffix array. Their makeshift alternatives require much resource than suffix arrays and consequently reduce practical performance of the compressed suffix array. Therefore, compressed suffix arrays still remain to be researched to apply to practical conditions.

References

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

[2] E. M. McCreight, "A space-economical suffix tree construction algorithm," J. ACM., Vol.23, No.2, pp.262-272, 1976.

[3] E. Ukkonen, "On-line construction of suffix trees," Algorithmica, Vol.14, pp.249-260, 1995.

[4] M. Farach, "Optimal suffix tree construction with large alphabets," Proc. 38th IEEE Symp. Found. Computer Science pp.137-143, 1997.

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

[6] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," SIAM J. Comput., Vol.22, No.5, pp.935-948, 1993.

[7] D. Gusfield, "An 'Increment-by-one' approach to suffix arrays and trees," Report. CSE-90-39, Computer Science Division, University of California, Davis, 1990.

[8] D. K. Kim, J. S. Sim, H. Park and K. Park, "Linear-time construction of suffix arrays," Proc. 14th Symp. Combinatorial Pattern Matching, pp.186-199, 2003.

[9] P. Ko and S. Aluru, "Space-efficient linear time construction of suffix arrays," Proc. 14th Symp. Combinatorial Pattern Matching, pp.200-210, 2003.

[10] J. Kärkkäinen and P. Sanders, "Simple linear work suffix array construction," Proc. 30th Int. Colloq. Automata Languages and Programming, pp.943-955, 2003.

[11] D. K. Kim, J. Jo and H. Park, "A fast algorithm for constructing suffix arrays for fixed-size alphabets," Proc. 3rd Int. Workshop on Experimental and Efficient Algorithms, pp.301-314, 2004.

[12] N. J. Larsson and K. Sadakane, "Faster Suffix Sorting," Report. LU-CS-TR:99-214, Dept. of Computer Science, Lund University, Sweden, 1999.

[13] G. Manzini and P. Ferragina, "Engineering a lightweight suffix array construction algorithm," Algorithmica, Vol.40, pp.33-50, 2004.

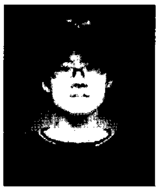
[14] K. Sch¨urmann and J. Stoye, "An incomplex algorithm for fast suffix array construction," Software: Practices and Experience, 2006 (to appear).

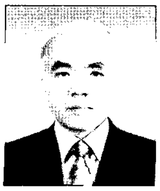
[15] J. I. Munro, V. Raman and S. S. Rao, "Space efficient suffix trees," J. of Algorithms, Vol.39, pp.205-222, 2001.

[16] R. Grossi and J. Vitter, "Compressed suffix arrays and suffix trees with applications to text indexing and string matching," Proc. 32nd ACM Symp. Theory of Computing, pp.397-406, 2000.

[17] P. Ferragina and G. Manzini, "Opportunistic data

structures with applications," Proc. 41st IEEE Symp. Found. Computer Science, pp.390-398, 2001.
[18] W. K. Hon, K. Sadakane and W. K. Sung, "Breaking a time-and-space barrier in constructing full-text indices," Proc. 44th IEEE Symp. Found. Computer Science, pp.251-260, 2003.
[19] J. C. Na, "Linear-time construction of compressed suffix arrays using $O(n \log^{\varepsilon} n)$-bit working space for large alphabets," Proc. 16th Combinatorial Pattern Matching, pp.57-67, 2005.
[20] R.M. Karp, R. E. Miller and A. L. Rosenberg, "Rapid identification of repeated patterns in strings," Proc. 4th ACM Symp. Theory of Computing, pp.125-136, 1972.
[21] H. Itoh and H. Tanaka, "An efficient method for in memory construction of suffix array," Proc. 11th Symp. String Processing and Information Retrieval, pp.81-88, 1999.
[22] D. K. Kim, J. C. Na, J. E. Kim and K. Park, "Efficient implementation of rank and select functions for succinct representation," Proc. 4th Int. Workshop on Experimental and Efficient Algorithms, pp.315-327, 2005.
[23] T. Hagerup, P. B. Miltersen and R. Pagh, "Deterministic dictionaries," J. of Algorithms, Vol.41, No.1, pp.69-85, 2001.
[24] D. E. Willard, "Log-logarithmic worst-case range queries are possible in space $\Theta(N)$," Information Processing Letters, Vol.17, No.2, pp.81-84, 1983.
[25] M. G. Maaß, "Matching statistics: efficient computation and a new practical algorithm for the multiple common substring problem," Software: Practices and Experience, Vol.36, No.3, pp.305-331, 2006.


박 치 성
2005년 2월 부산대학교 정보컴퓨터공학과(공학사). 2007년 2월 부산대학교 컴퓨터공학과(공학석사). 2007년 3월~현재 삼성전자 정보통신총괄 사원. 관심분야는 Suffix Tree, Suffix Array, Compressed Suffix Array, Cryptology


김 민 환
1980년 2월 서울대학교 전기공학과 학사 1983년 2월 서울대학교 컴퓨터공학과 석사. 1988년 8월 서울대학교 컴퓨터공학과 박사. 1986년 3월~현재 부산대학교 컴퓨터공학과 교수. 관심분야는 영상분석 및 컴퓨터시각

이 석 환
1999년 경북대학교 전자공학과 졸업(공학사). 2001년 경북대학교 전자공학과 졸업(공학석사). 2004년 경북대학교 전자공학과 졸업(공학박사). 2005년~현재 동명대학교 정보보호학과 조교수. 관심분야는 워터마킹, DRM, 영상신호처리

권 기 룡
1986년 2월 경북대학교 전자공학과 졸업(공학사). 1990년 2월 경북대학교 대학원 전자공학과 졸업(공학석사). 1994년 8월 경북대학교 대학원 전자공학과 졸업(공학박사). 2000년 7월~2001년 8월 Univ. of Minnesota, Post-Doc. 과정. 1996년 3월~2006년 2월 부산외국어대학교 디지털정보공학부 부교수. 2006년 3월~현재 부경대학교 전자컴퓨터정보통신공학부 교수. 2005년 1월~현재 한국멀티미디어학회 논문지편집 위원장. 관심분야는 멀티미디어 정보보호, 멀티미디어 영상처리, 웨이브릿 변환

김 동 규
1992년 2월 서울대학교 컴퓨터공학과(공학사). 1994년 2월 서울대학교 컴퓨터공학과(공학석사). 1999년 2월 서울대학교 컴퓨터공학과(공학박사). 1999년 9월~2006년 2월 부산대학교 컴퓨터공학과(조교수). 2006년 3월~현재 한양대학교 전자통신컴퓨터공학과(부교수). 관심분야는 Cryptology, 암호화 모듈 칩 설계, String processing