압축된 써픽스 배열을 직접 구축하는 선형시간 알고리즘

성종희 전정은 김동규[0]
부산대학교 컴퓨터공학과
{jhsung, jejun, dkkim[0]}@islab.ce.pusan.ac.kr

Direct Construction Algorithms for Compressed Suffix Arrays in Linear Time

Jong Hi Sung[0] Jung Eun Jun Dong Kyue Kim
School of Electrical and Computer Engineering, Pusan National University

요 약

써픽스 배열은 써픽스 트리와 더불어 바이오인포매틱스(bioinformatics) 등에 널리 사용되는 전체 텍스트(full-text)의 인덱스 자료구조이다. 여러 응용분야에서 처리해야하는 데이터양의 기하급수적인 증가에 따라, 써픽스 배열을 압축하여 저장해야 하는 필요성이 커지고 있다. Grossi와 Vitter는 주어진 스트링의 써픽스 배열이 있을 경우, 작은 저장 공간을 사용하는 압축된 써픽스 배열(compressed suffix arrays)을 정의하였다. 본 논문에서는 주어진 스트링에서 써픽스 배열을 구축할 필요 없이, 직접적으로 압축된 써픽스 배열을 구축하는 선형시간 알고리즘을 제시한다.

1. 서론

바이오인포매틱스의 발전이 생물학 데이타베이스의 증가를 가져옴에 따라, 대용량의 전체 텍스트에 대해서 빠른 검색을 지원해 주는 인덱스 자료 구조의 필요성이 대두되었다. 인덱스 자료 구조의 대표적인 것으로는 써픽스 트리(suffix tree)[1, 2, 3, 4, 5, 6, 7]와, 써픽스 배열(suffix array)[8, 9]이 있다.

써픽스 배열은 써픽스 트리에 비해 구조가 간단하고, 저장 공간을 적게 차지한다는 점에 있어서 매우 유용하다. 지금까지 써픽스 배열은 구축 시간 면에서 써픽스 트리에 비해 불리하였으나, 최근에 써픽스 배열을 선형 시간에 구축하는 알고리즘[10]으로 인해서 이론적인 시간 복잡도가 동등하게 되었다.

길이가 n인 스트링의 써픽스 배열은 $O(n\log n)$ 비트의 저장 공간을 사용한다. 문자 집합 Σ상의 스트링이 $O(n\log|\Sigma|)$ 비트의 공간을 사용함에 비해 $\Omega(\log_{|\Sigma|} n)$배나 많으므로, 대용량의 인덱스 자료구조로는 적합하지 못하다. 그래서 인덱스 자료 구조의 공간을 줄이는 문제에 대한 연구[11, 12]들이 행해졌다. 그 중 Grossi와 Vitter의 연구는 주어진 스트링의 써픽스 배열이 있을 경우, 작은 저장 공간을 사용하는 압축된 써픽스 배열(compressed suffix arrays)을 구축하는 알고리즘을 제시했다.

본 논문은 스트링에서 직접 압축된 써픽스 배열을 만드는 알고리즘을 제시한다. 이 알고리즘은 써픽스 배열을 구축하는 선형시간 알고리즘을 참고하여, Σ의 크기가 고정된 문자집합상의 스트링에 대해서 압축된 써픽스 배열을 선형시간에 구축한다.

본 논문의 구성은 다음과 같다. 2장에서는 본 논문에서 사용되는 기본적인 용어들을 정의하고, 3장에서는 써픽스 배열을 구축하는 알고리즘에 대해 살펴본다. 4장에서는 압축된 써픽스 배열을 직접 구축하는 알고리즘을 제시하며, 5장에서 결론을 맺는다.

2. 기본 정의

2.1 써픽스 배열

써픽스 배열 SA는 Pos와 Lcp의 두 배열 형식의 자료 구조로 구성된다. Pos는 다음과 같이 정의한다. 길이가 n인 스트링 T가 존재할 때, T의 써픽스들을 사전적 순서(lexicographical order)에 따라 정렬시킨 후, 그 순서를 배열형태로 가지고 있는 자료 구조이다. Lcp는 Pos에서 서로 인접해 있는 두 써픽스들 간에, 가장 길게 매칭(matching)되는 프리픽스(prefix)의 길이를 배열로 저장해 놓은 자료 구조이다. [그림1]의 (a)는 써픽스 배열의 예를 보여주고 있다. 본 논문에서 SA는 Lcp는 고려하지 않고, Pos만을 의미한다.

2.2 압축된 써픽스 배열

Grossi와 Vitter의 연구에서 제시된 압축된 써픽스 배열은 Pos만을 고려한다. Lcp 압축은 Sadakane[13]가 제시한 방식을 이용한다. 써픽스 배열을 압축하는 방식에는 두 가지가 있는데, 두 압축 방식은 차지하는 공간의 크기가 다르고, 압축된 써픽스 배열에서 본래의 써픽스 배열의 정보를 찾는데(*lookup*) 걸리는 시간이 다르다.

Grossi와 Vitter가 제시하는 알고리즘은 길이가 n인 스트링 T의 써픽스 배열 SA를 입력으로 하여 다음과 같이 처리한다. SA를 0번째 단계의 써픽스 배열 SA_0로 두면, 길이 n_0는 $n_0 = n$ 이 된다. 1번째 단계의 써픽스 배열 SA_1은 SA_0에 저장되어 있는 짝수 써픽스를 2로 나누어 저장한 것으로 길이 $n_1 = n/2$ 이다. 이 과정을 반복하면, $k \geq 0$인 단계 k 에서의 써픽스 배열은 SA_k이고, 그 길이 n_k는 $n_k = n/2^k$ 이 된다. 이 작업을 마지막 단계 $l = \log\log_{|\Sigma|} n$까지 반복하여 수행한다.

써픽스 배열은 다음과 같은 방법으로 압축된다. 마지막 단계 l에서의 써픽스 배열인 SA_l을 제외한 $0 \leq k < l$인 단계 k의 써픽스 배열 SA_k는 저장되지 않는다. 이를 대신해서 두 가지 압축 표현 방식에 따라, 각각 다른 자료 구조들을 저장한다.

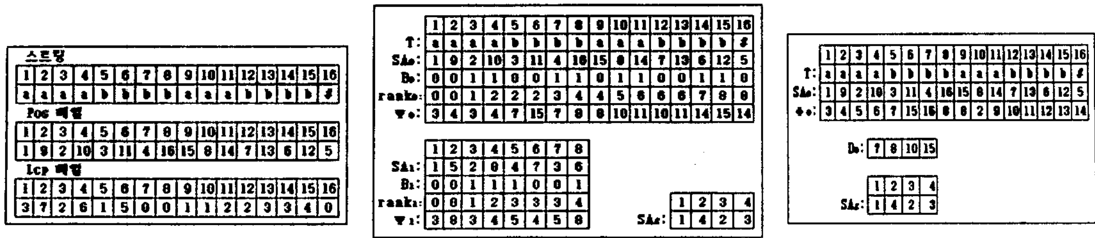
① $(1+\frac{1}{2}\log\log_{|\Sigma|} n)n\log|\Sigma| + O(n)$ 비트로 압축하는 방식

첫 번째 써픽스 배열 압축 방식은 $0 \leq k < l$인 단계 k마다 SA_k 대신 B_k, Ψ_k, $rank_k$ 값을 저장한다. $1 \leq i \leq n_k$인 i에 대해서 $B_k[i]$, $\Psi_k[i]$, $rank_k[i]$의 값을 다음과 같이 정의한다:

- $SA_k[i]$가 짝수일 때, $B_k[i] = 1$; 그렇지 않으면, $B_k[i] = 0$.
- $SA_k[i]$가 홀수이고 $SA_k[j] = SA_k[i]+1$일 때, $\Psi_k[i] = j$; 그렇지 않으면, $\Psi_k[i] = i$.
- $rank_k[i]$는 B_k 배열에서 i번째 인덱스까지 1의 갯수.

[그림1]의 (b)는 (a)에 나타난 써픽스 배열을 첫 번째 압축 방식을 사용하여 압축한 예를 보여주고 있다. 이 때, l은 $\log\log_2 16 = 2$가 되고, 2를 제외한 모든 단계에 대해서, B_k, $rank_k$, Ψ_k 값이 정의에 따라 계산된 것을 볼 수 있다.

첫 번째 압축 방식은 $(1+\frac{1}{2}\log\log_{|\Sigma|} n)n\log|\Sigma| + O(n)$ 비트의 공간을 차지하고, 구축하는데 걸리는 시간은 $O(n\log|\Sigma|)$이다. 그리고 압축된 써픽스 배열에서 $1 \leq i \leq n$인 i에 대해서 i번째 써픽스 배열을 찾는 함수인 *lookup(i)*가 수행되는 시간은 $O(\log\log_{|\Sigma|} n)$이다. $|\Sigma| = 2$이면 l은 $\log\log n$이 되고, $\frac{1}{2} n\log\log n + O(n)$ 비트의 공간을 차지하며, 구축 시간은 $O(n)$, *lookup(i)*가 수행되는 시간은 $O(\log\log n)$이다.



(a) 써픽스 배열

(b) $(1+\frac{1}{2}\log\log_{|\Sigma|}n)n\log|\Sigma|+O(n)$

비트로 압축된 써픽스 배열의 예

(c)

$(1+\varepsilon^{-1})n\log|\Sigma|+o(n\log|\Sigma|)$

비트로 압축된 써픽스 배열의 예

그림 1. 써픽스 배열과 압축된 써픽스 배열

② $(1+\varepsilon^{-1})n\log|\Sigma|+o(n\log|\Sigma|)$ 비트로 압축하는 방식

두 번째 압축 방식은 $k\in\{0, \varepsilon l, 2\varepsilon l, 3\varepsilon l, \cdots, (1-\varepsilon)l\}$인 단계 k마다 SA_k 대신 Φ_k, D_k 값을 저장한다. Φ_k, D_k는 다음과 같이 정의한다:

- $1\le i\le n_k$인 i에 대해서 $SA_k[i]\ne n_k$, $SA_k[j]=SA_k[i]+1$이면, $\Phi_k[i]=j$; 그렇지 않으면, $\Phi_k[i]=i$.
- k의 다음 단계를 k'라고 할 때, $1\le i\le n_{k'}$인 i에 대해서 $D_k[i]$에는 $SA_{k'}$의 원소들과 연관된 SA_k의 인덱스가 저장되어 있다.

[그림1]의 (c)는 두 번째 압축 방식의 예를 보이고 있는데, Φ_k, D_k 값이 정의에 따라 계산된 것을 볼 수 있다. 예에서 D_0의 값은 SA_2의 원소들과 연관된 SA_0의 인덱스를 저장하고 있다. SA_2의 원소들과 연관된 써픽스들은 SA_0에서 $<4, 16, 8, 12>$이고, D_0는 SA_0에서 이 써픽스들의 인덱스 $<7, 8, 10, 15>$를 저장한다.

이 방식은 $(1+\varepsilon^{-1})n\log|\Sigma|+o(n\log|\Sigma|)$ 비트의 공간을 차지하고, 구축 시간은 $O(n\log|\Sigma|)$, *lookup(i)*가 수행되는 시간은 $O(\log^{\varepsilon}_{|\Sigma|}n)$이다. $|\Sigma|=2$이면 l은 $\log\log n$이 되고, $(1+\varepsilon^{-1})n+O(n/\log\log n)$ 비트의 공간을 차지하며, 구축 시간은 $O(n)$, *lookup(i)*가 수행되는 시간은 $O(\log^{\varepsilon}n)$이다.

3. 써픽스 배열을 구축하는 알고리즘

Manber와 Myers[8]가 제시한 최초의 써픽스 배열 구축 알고리즘은 구축 시간이 $O(n\log n)$이다. 이 알고리즘은 *Pos*를 구하기 위해 더블링(doubling)기법을 사용한다. 그리고 *Lcp*를 구하기 위해 완전 균형 이진 트리(balanced full binary tree)를 유지한다.

Gusfield의 연구[9]에서 제시한 써픽스 배열 구축 알고리즘은 Manber와 Myers가 제시한 알고리즘과 같은 시간이 소요된다. *Pos*는 하나씩 증가하는 접근 방식(increment-by-one approach)을 통해서 구해진다. 또한 Manber와 Myers의 알고리즘에 비해, $O(n)$시간 내에 *Lcp*를 간단하게 구해 낼 수 있다.

최근에 연구된 써픽스 배열을 구축하는 선형 시간 알고리즘[10]은 정수 문자집합상의 스트링에 대해서 써픽스 배열을 구축한다. 이 알고리즘은 가장 활발히 연구되고 있는 구축 방식인 논문 [2, 3, 4, 7]에서 사용하고 있는 재귀적인 분할 정복 기법(divide-and-conquer approach)을 따르고 있다. 길이가 n인 스트링 T의 써픽스 배열 구축 과정은 다음의 3 단계로 이루어진다.

- Step 1. 스트링의 홀수 번째 써픽스들의 배열인 *홀수 배열* SA_O를 재귀적으로 구축한다.
- Step 2. SA_O로부터 스트링의 짝수 번째 써픽스들의 배열인 *짝수 배열* SA_e를 구축한다.
- Step 3. SA_O와 SA_e를 합쳐서 T의 써픽스 배열 SA_T를 만든다.

Step.1 홀수 배열 구축: SA_O는 선형 시간 내에 다음과 같이 구축된다. T를 $1\le i\le n/2$인 모든 i에 대해서 $(T[2i-1], T[2i])$로 표현할 수 있다. 이것을 $[1, n/2]$ 사이에 있는 수에 연결시키면 $\frac{n}{2}$ 길이의 새로운 스트링 T'로 인코딩(encoding)된다. 재귀적으로 이 과정을 수행하여, T'의 써픽스 배열 $SA_{T'}$를 만든다. $SA_{T'}$로부터 SA_O를 얻어낸다.

Step.2 짝수 배열 구축: SA_e는 SA_O로부터 선형 시간 내에 만들어진다. 홀수 번째 써픽스를 S_{2i+1}으로 나타내면, 짝수 번째 써픽스는 $(T[2i], S_{2i+1})$으로 표현된다. 두 번째 요소는 홀수 배열 구축 과정을 통해 정렬되어 있으므로, 첫 번째 요소를 가지고 다시 정렬을 수행하여 SA_e를 얻는다.

Step.3 홀수 배열과 짝수 배열 합치기: SA_O와 SA_e를 합쳐서 선형 시간 내에 SA_T를 만든다. 이 단계는 가장 복잡한 단계이므로, 지면상 설명은 생략한다.

4. 알고리즘

본 논문에서 제시하는 알고리즘은 문자집합 Σ를 고정된 크기로 간주하여, 길이가 n인 스트링 T의 압축된 써픽스 배열을 구축한다. 이 알고리즘은 써픽스 배열을 구축하는 선형 시간 알고리즘과 유사하다.

본 논문에서 제시하는 알고리즘은 SA_e를 반복적으로 구축하고, SA_O와 SA_e를 합치는 과정에서 압축을 수행한다. 압축 방식은 Grossi와 Vitter가 제시한 두 가지 압축 방식을 모두 적용할 수 있다. 그러므로, 기존의 알고리즘과 동일하게 저장 공간을 사용하면서 압축된 써픽스 배열을 구축할 수 있다.

4.1 $(1+\frac{1}{2}\log\log_{|\Sigma|}n)n\log|\Sigma|+O(n)$ 비트로 압축된 써픽스 배열

① 짝수 배열 구축: SA_e는 다음과 같이 구축한다. T를 $1\le i\le n/2$인 모든 i에 대해 $(T[2i], T[2i+1])$으로 나타내어, $\frac{n}{2}$길이의 스트링 T'로 만든다. 이 과정을 $k\ge0$인 단계 k를 사용해서 나타내면, $k=0$일 때 스트링의 길이 n_0는 n으로 시작하여 마지막 단계 $l=\log\log_{|\Sigma|}n$까지 이 과정을 반복한다. 단계 l에서 T의 길이 n_k는 $\frac{n}{2^{\log\log_{|\Sigma|}n}}=\frac{n}{\log_{|\Sigma|}n}$이 된다. 이 때, 스트링 T'에 대해서 써픽스 배열을 $O(n\log n)$시간에 구축하는 알고리즘을 적용하여, $SA_{T'}$를 만든다. SA_e는 $SA_e[i]=2SA_{T'}[i]$로 얻고, 그 길이는 n_e로 표현한다.

② 홀수 배열 구축: 홀수 번째 써픽스들을 $(T[2i-1], S_{2i})$로 표현하면, 두 번째 요소는 짝수 배열 구축 과정에서 정렬되어 있다. 따라서 첫 번째 요소에 대해 다시 정렬을 수행하여 SA_O를 구축하고, 그

길이는 n_O로 표현한다.

Ψ_k는 Grossi와 Vitter가 제시한 정의에 따라, 각 홀수 써픽스들에 대해, 스트링에서 해당 홀수 써픽스 다음 위치에 존재하는 짝수 써픽스를 저장하는 전체 써픽스 배열의 인덱스를 저장한다. SA_O 구축 시 홀수 써픽스와 연관된 짝수 써픽스를 저장하는 SA_e의 인덱스를 알 수 있으므로, Ψ_k를 구해주기 위해서 홀수 배열 구축 과정에서 전 처리 작업을 수행하는 것이 필요하다. $O_k[i]$는 스트링에서 $SA_O[i]$가 나타내는 홀수 써픽스 다음 위치의 짝수 써픽스를 저장하는 SA_e의 인덱스를 가리킨다. SA_O 구축 후, 다음과 같이 $O_k[i]$ 값을 구한다:

- $1 \le i \le n_o$인 i에 대해서 $SA_e[j] = SA_o[i]+1$이면, $O_k[i] = j$.

$O_k[i]$는 SA_e의 인덱스이므로, SA_T의 인덱스로 바뀌어 져야 한다. $1 \le i \le n_e$인 i에 대해서 $rptr_k[i]$는 SA_T가 구축된 후에, O_k의 값을 SA_T의 인덱스로 바꿔 주기 위해서 O_k의 인덱스를 저장하고 있는 역포인터(reverse pointer)로 정의하고, 다음과 같이 구한다:

- $1 \le i \le n_e$인 i에 대해서 $O_k[j] = i$이면, $rptr_k[i] = j$.

③ 짝수 배열과 홀수 배열 합치면서 압축하기: SA_e와 SA_O를 합치는 작업은, $1 \le i \le n_e$인 모든 i에 대해 $SA_e[i]$를 저장하는 SA_T의 인덱스와, $1 \le i \le n_O$인 모든 i에 대해서 $SA_O[i]$를 저장하는 SA_T의 인덱스를 찾는 문제로 생각할 수 있다[10]. $ind_e[i]$와 $ind_O[i]$는 각각 이 인덱스를 가리키고, 다음과 같이 정의한다:

- $1 \le i \le n_e$인 i에 대해서 $SA_T[j] = SA_e[i]$이면 $ind_e[i] = j$.
- $1 \le i \le n_O$인 i에 대해서 $SA_T[j] = SA_O[i]$이면 $ind_O[i] = j$.

$ind_e[i]$와 $ind_O[i]$는 써픽스 배열을 선형 시간에 구축하는 알고리즘에 따라 SA_T를 구축할 때 구할 수 있다.

써픽스 배열을 압축하기 위해서 마지막 단계의 써픽스 배열 SA_l만을 저장한다. 그리고 B_k, Ψ_k 값을 다음과 같이 구한다:

- $ind_e[i]$를 구하면, $B_k[ind_e[i]] = 1$, $O_k[rptr_k[i]] = ind_e[i]$, $\Psi_k[ind_e[i]] = ind_e[i]$.
- $ind_O[i]$를 구하면, $B_k[ind_O[i]] = 0$.
- $1 \le i \le n_O$인 모든 i에 대해서 $ind_O[i]$값을 구하면, $\Psi_k[ind_O[i]] = O_k[i]$.

$rank_k$의 값은 SA_T가 생성된 후에, Grossi와 Vitter의 연구에서 제시한 방식대로 구한다. 위와 같이 ind_e, ind_O 값을 구하면서 B_k, $rank_k$, Ψ_k의 값을 정해줌으로써, SA_T를 구축하면서 압축을 수행할 수 있다.

4.2 $(1+\varepsilon^{-1})n\log|\Sigma| + o(n\log|\Sigma|)$ 비트로 압축된 써픽스 배열

써픽스 배열을 구축하는 방식은 4.1에서 설명한 것과 같다.

두 번째 압축 방식은 $k \in \{0, \varepsilon l, 2\varepsilon l, 3\varepsilon l, \cdots, (1-\varepsilon)l\}$인 단계 k마다 Φ_k와 D_k의 값을 구하고, 마지막 단계인 $\log\log_{|\Sigma|} n$ 단계의 써픽스 배열을 저장한다. D_k의 값은 Grossi와 Vitter의 연구에서 제시한 방식대로 구한다.

두 번째 압축 방식은 Φ_k의 정의에 따라, 짝수 번째 써픽스들에 대해서도 O_k와 유사한 기능을 수행하는 O'_k를 정의한다. $O'_k[i]$는 스트링에서 $SA_e[i]$가 나타내는 짝수 써픽스 다음 위치의 홀수 써픽스를 저장하는 SA_O의 인덱스를 가리킨다. O'_k를 유지하기 위해서는 홀수 번째 써픽스들이 정렬되어 있어야 하므로, 홀수 배열이 구축된 후에 다음과 같이 O'_k의 값을 구한다:

- $1 \le i \le n_e$인 i에 대해서 $SA_O[j] = SA_e[i]+1$이면, $O'_k[i] = j$.

O'_k값 역시 홀수 배열의 인덱스를 가지고 있으므로, SA_T를 구축한 후에 SA_T의 인덱스로 바꾸어준다. 그리고 다음과 같이 Φ_k의 값을 구한다:

- $SA_T[ind_e[i]] \ne n_k$이면, $\Phi_k[ind_e[i]] = O'_k[i]$.
- $SA_T[ind_O[i]] \ne n_k$이면, $\Phi_k[ind_O[i]] = O_k[i]$.

정리 1. Σ가 고정된 문자집합상의 스트링 T에 대한 압축된 써픽스 배열을 구축하는 데 걸리는 시간은 $O(n)$이다.

증명. 첫 번째 압축 방식으로 압축된 써픽스 배열을 구축하는 시간은 다음과 같다:

(짝수 배열 구축 시간) + (홀수 배열 구축 시간) + (짝수 배열과 홀수 배열을 합치는 시간) + (B_k, $rank_k$, Ψ_k의 값을 구하는 시간)

$$= O(n\log|\Sigma|) + O(n) + O(n) + \sum_{k=0}^{\log\log_{|\Sigma|} n-1} O(n_k) = O(n\log|\Sigma|).$$

두 번째 압축 방식으로 압축된 써픽스 배열을 구축하는 시간은 첫 번째 방식과 Φ_k와 D_k를 구하는 시간만이 다르다. 마지막 단계 l은 $\log\log_{|\Sigma|} n$이고, 총 구축 시간은 다음과 같다:

$$O(n\log|\Sigma|) + O(n) + O(n) + \sum_{k=i\varepsilon l,\ 0 \le i \le \varepsilon^{-1}} O(n_k) = O(n\log|\Sigma|).$$

즉, 스트링에서 압축된 써픽스 배열을 구축하는 데 걸리는 시간은 $O(n\log|\Sigma|)$시간이다. Σ가 고정된 문자집합일 경우에는 스트링에서 압축된 써픽스 배열을 구축하는 데 걸리는 시간은 $O(n)$이다. □

5. 결론

본 논문에서는 스트링에서 직접적으로 압축된 써픽스 배열을 구축하는 알고리즘을 제시했다. 이 알고리즘을 Σ상에서 정의된 스트링에 적용할 경우, 총 구축시간이 $O(n\log|\Sigma|)$시간 걸린다. 본 논문에서는 $|\Sigma|$가 상수라고 가정하였으므로, 선형 시간 내에 압축된 써픽스 배열을 구축할 수 있다. 본 논문에서 제시하는 알고리즘은 c를 상수라고 가정하였을 경우, $|\Sigma| = n^c$인 경우에도 확장이 가능하다.

6. 참고 문헌

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

[2] M.Farach, Optimal suffix tree construction with large alphabets, IEEE symp. Found. Computer Science, 137-143, 1997

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

[4] S.C.Sahinalp and U.Vishkin, Symmetry breaking for suffix tree construction, IEEE Symp. Found. Computer Science, 300-309, 1994

[5] E.Ukkonen, On-line construction of suffix trees, Algorithmica 14, 249-260, 1995

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

[7] M.Farach and S.Muthukrishnan, Optimal logarithmic time randomized suffix tree construction, Proc.International Colloquium on Automata Languages and Programming, 550-561, 1996

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

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

[10] D.K.Kim, J.S.Sim, H.Park and K.Park, Linear-time construction of suffix arrays, accepted to Combinatorial Pattern Matching, 2003

[11] J.C.Na, A.Apostolico, C.S.Iliopoulos and K.Park, Truncated suffix trees and their application to data compression, to appear Theoretical Computer Science.

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

[13] K.Sadakane, Succinct representation of lcp information and improvement in the compressed suffix arrays, ACM-SIAM Symp. on Discrete Algorithms, 225-232, 2002