문자의 빈도수를 고려한 *Rank/Select* 자료구조 구현

(Implementation of *Rank/Select* Data Structure using Alphabet Frequency)

권 유 진 † 이 선 호 †† 박 근 수 †††
(Yoojin Kwon) (Sunho Lee) (Kunsoo Park)

요 약 *Rank/select* 자료구조는 트리, 그래프, 문자열 인덱스 등의 다양한 자료구조를 간결하게 표현하는 기본 도구이다. *Rank/select* 자료구조는 주어진 문자열에 어느 위치까지 나타난 문자 개수를 세는 연산을 처리한다. 효율적인 *rank/select* 자료구조를 위해 이론적인 압축 방식들이 제안되었으나, 실제 구현에 있어 연산 시간 및 저장 공간의 효율을 보장할 수 없었다. 본 논문은 간단한 방법으로 이론적인 압축 크기를 보장하면서 연산 시간도 효율적인 *rank/select* 자료구조 구현 방법을 제시한다. 본 논문의 실험을 통해, 복잡한 인코딩 방법 없이도 이론적인 $nH_0 + O(n)$ 비트 크기에 근접하면서 기존의 HSS 자료구조보다 빠른 *rank/select* 연산을 지원하는 구현 방법임을 보인다.

키워드 : *rank/select* 자료구조, 인덱스 자료구조, 간결한 자료구조

Abstract The *rank/select* data structure is a basic tool of succinct representations for several data structures such as trees, graphs and text indexes. For a given string sequence, it is used to answer the occurrence of characters up to a certain position. In previous studies, theoretical *rank/select* data structures were proposed, but they didn't support practical operational time and space. In this paper, we propose a simple solution for implementing *rank/select* data structures efficiently. According to experiments, our methods without complex encodings achieve $nH_0 + O(n)$ bits of theoretical size and perform *rank/select* operations faster than the original HSS data structure.

Key words : *rank/select* data structures, index data structures, succinct data structures

1. 서 론

Rank/select 자료구조는 주어진 문자열을 미리 처리해서 문자열에 나타난 문자 개수를 세는 간단한 연산을 계산하는 자료구조이다. 자료구조가 계산하는 *rank/select* 연산은 다음과 같이 정의한다.

- *rank*(c, i): 문자 c 가 위치 i 까지 나타난 개수
- *select*(c, k): 문자 c 가 k 번째로 나타난 위치

예 1. 문자열 T = acabbc 일 때 *rank*(a, 5) = 2 즉, 5번째 위치까지 나타난 a의 개수인 2를 반환하고 *select*(a, 2) = 3 즉, 2번째로 나타난 a가 문자열 내에 있는 위치인 3을 반환한다.

이러한 *Rank/select* 자료구조는 간단한 연산을 지원하지만 이를 이용하면 문자열의 패턴 검색과 같은 복잡한 기능을 지원하는 인덱스 자료구조를 만들 수 있다. 문자열의 문자 집합을 A, 길이를 n이라고 할 때, 기존의 인덱스 자료구조인 접미사 트리[1]나 접미사 배열 [2]은 그 크기가 $O(n\log n)$ 비트로, 문자열 자체의 크기 $n\log|A|$ 비트보다 커지는 문제점을 갖고 있다. 이를 해결하기 위해 압축 인덱스 자료구조[3,4]가 제안되었으며, 여기서 *rank/select* 자료구조가 중요한 역할을 담당한다[4,5].

원래 *rank/select* 자료구조에 대한 연구는 비트열에

† 정 회 원 : 한국IBM 유비쿼터스컴퓨팅랩
yjkwon@theory.snu.ac.kr
†† 정 회 원 : 한양대학교 전기통신컴퓨터공학부
shlee@theory.snu.ac.kr
††† 종신회원 : 서울대학교 컴퓨터공학부 교수
kpark@theory.snu.ac.kr
논문접수 : 2008년 12월 25일
심사완료 : 2009년 5월 7일

정보과학회논문지: 시스템 및 이론 제36권 제4호(2009.8)

대한 *rank/select* 연산을 계산하는 자료구조[6-8]로부터 출발했다. 비트 *rank/select* 연산을 이용하면 트리나 그래프와 같은 복잡한 자료구조를 압축한 형태로 표현할 수 있기 때문이며[8], 일반적인 문자열에 대한 *rank/ select* 연산도 비트 *rank/select* 연산을 이용하여 계산할 수 있다[9]. 가장 간단한 비트 *rank/select* 자료구조의 구성 방법은 위치 i까지 나타난 비트 1의 개수나 k번째 비트 1의 위치를 모두 테이블 형태로 저장해 두는 방식인데, 물론 이 방식은 저장 공간의 크기가 너무 커지게 되므로 제안된 방법들은 비트열을 적당한 크기의 블록으로 쪼개서 일부 비트 1의 개수 및 위치 정보를 기록해두고 나머지 정보는 비트열을 참조하여 계산한다.

일반적인 문자열에 대해서도 여러 가지 이론적인 *rank/select* 자료구조가 제안되었으나[9-11], 실제 상황에서 쉽게 구현할 수 있는 방법은 크게 두 가지가 있다[12]. 하나는 wavelet 트리[9]를 이용하여 문자 c에 대한 *rank/select* 연산을 $\log|A|$개의 비트 *rank/select* 연산으로 변환하여 처리하는 방법이다. 다른 하나는 비트 *rank/select*처럼 문자열을 적당한 크기의 블록으로 쪼개서 블록 시작 위치까지 나타난 문자 개수와 블록 내에서 문자 위치(오프셋)를 기록해두고 *rank/select* 연산을 계산하는 HSS 방식이다[10]. HSS는 처음 Compressed Suffix Array를 만들 때 사용된 자료구조로[10], Hon, Sung, Sadakane의 이름 머릿글자를 차용해 본 논문에서 이름 붙인 것이다. 이들의 자료구조가 *rank/select* 문제에 적합한 자료구조의 형태로 변형되고, 압축 방법이 추가되어 최근 논문에서 제안되었다[11].

본 논문에서는 압축 HSS *rank/select* 자료구조[11]를 구현하기 위해, 기존의 복잡한 인코딩에 의존하지 않는 간단한 자료구조를 제안한다. 자료구조 생성단계에서 Huffman 코딩 혹은 Shannon 코딩을 이용하여 블록 크기를 조절해두면, *rank/select* 연산을 처리하는 과정에서는 인코딩/디코딩 처리가 필요 없다. 따라서 이론적인 $nH_0 + O(n)$ 비트의 공간을 얻으면서도 연산 처리 속도는 압축하지 않은 HSS 자료구조와 비슷한 구현 방법을 얻을 수 있다. 또한 본 논문에서는 Wavelet 트리를 이용한 방법은 고려하지 않았다. HSS 자료구조가 *rank/select* 연산을 위해 $O(1)$ 횟수의 비트 *rank/select* 연산을 수행하는 반면, Wavelet 트리는 $O(\log|A|)$ 횟수의 비트 *rank/select* 연산을 수행한다(표 1). 현재 알려진 비트 *rank/select* 구현 방법이 충분히 빠르지 않기 때문에 연산 처리 과정에서 병목 부분은 비트 *rank/select* 연산이다. 따라서, 특히 문자 집합 크기가 커질수록, Wavelet 트리를 이용한 구현 방법은 HSS 방식에 비해 처리 속도가 느려진다.

2. HSS *rank/select* 자료구조 소개

HSS 자료구조는 전체 길이 n의 문자열을 문자 집합 크기 $|A|$의 블록들로 쪼갠 뒤 0과 1로 이루어진 2개의 비트열(bit array)에 다음과 같은 정보를 저장한다. 블록에 대한 정보는 B라는 배열에, 나머지의 정보는 R이라는 배열에 저장한다.

HSS는 문자열을 $|A|$크기의 블록으로 쪼개서 총 $n/|A|$개의 블록을 구하고, 각 문자 c에 대해 블록 내에 나타난 개수를 unary code로 기록한 비트열 B_c를 구한다. 예를 들면 문자 c가 첫 번째 블록에 3개, 두 번째 블록에 2번 나왔으면 B_c=1000100...과 같이 나타낸다. 비트열 B_c들을 전부 붙여서 전체 비트열 B를 만들어두면, B에 대한 비트 *rank/select* 연산을 통해 블록 시작 위치까지 나타난 문자 개수를 계산할 수 있다.

R은 블록 내에서 각 문자의 위치(오프셋)를 저장하는데, 한 블록의 크기가 $|A|$이므로 오프셋의 길이는 $\log|A|$가 된다. 각 문자 c에 대한 오프셋은 배열 R_c에 B_c와 순서를 맞춰서 저장된다. k번째 문자 c는 B_c상에 k번째 0으로 기록되고 R_c상에는 k번째 위치에 자신이 속한 블록 내에서의 위치가 기록된다. 오프셋 배열 R_c들도 전부 붙여서 전체 오프셋 배열 R을 만들어둔다. R_c를 참조하여 블록 내의 특정 위치보다 작은 위치 값을 갖는 문자 c개수를 세면 최종 *rank* 연산을 계산할 수 있다. 비슷한 방법으로 *select* 연산도 B와 R을 참조하여 계산한다.

예 2. T_{20} = AGCTTGTGGTTATTTGTCGT의 문자열이 있다. 문자 빈도수(n_A = 2 , n_C = 2, n_G = 6, n_T

표 1 문자 집합 A에 대한 *rank/select* 자료구조

자료 구조	크기(bit)	비트 *rank/select* 횟수	전체 연산 시간	
			rank	*select*
$\|A\|$ 개의 비트열	$n\|A\| + o(n\|A\|)$	$O(1)$	$O(1)$	$O(1)$
Wavelet 트리	$n\log\|A\| + o(n\log\|A\|)$ 또는 $nH_0 + o(n\log\|A\|)$	$O(\log\|A\|)$	$O(\log\|A\|)$	$O(\log\|A\|)$
HSS	$n\log\|A\| + O(n)$ 또는 $nH_0 + O(n)$	$O(1)$	$O(\log\|A\|)^*$	$O(1)$

*HSS의 부가적인 binary search 과정은 비트 *rank/select* 연산에 비해 매우 빠름

블록1	블록2	블록3	블록4	블록5
A		A		
C				C
G	G G	G	G	G
T	T T	TT	TTT	T T

그림 1 HSS 자료구조를 위한 T_{20} = AGCTTGTGGTTATTTGTCGT의 블록 예제

= 10)는 모두 알고 있다. 위의 문자열은 길이 4 단위의 블록으로 쪼개져 총 5개의 블록으로 나뉘게 된다.

이들의 블록정보는 B배열에, 나머지 블록 내의 위치 정보는 R배열에 다음과 같이 저장된다.

B = 1011011 101110 10100101010 101001001000100
R = 0011 1001 010111001110 11001001100001100011

그리하여 HSS 자료구조의 크기는 다음의 R 크기와 B 크기의 총합인 $n\log|A|+O(n)$ 비트가 된다. 여기서 n_c는 문자 c의 빈도수, 즉 문자열에 나타난 총 횟수를 의미한다.

$$|B| = \Sigma_c(n_c + n/|A|) = 2n + O(|A|)$$
$$|R| = \Sigma_c\, n_c\log|A| = n\log|A| \qquad (1)$$

본 논문에서 해결하고자 하는 문제는 HSS 자료구조의 크기를 줄이는 것이다. HSS 계열의 자료구조 크기를 줄이기 위해 우리는 문자의 빈도수를 고려한 압축방식을 적용할 수 있다. 문자의 빈도수를 고려한 엔트로피 H_0는 다음과 같이 $H_0 = -\Sigma_c\ (n_c/n)\log(n_c/n)$로 정의하는데, 일반적으로 H_0는 $\log|A|$ 보다 작다[13]. R의 크기를 문자열 자체의 크기보다 작은 nH_0 비트로 줄이기 위해 gap 인코딩[14], block identifier 인코딩[15] 등의 방법이 제안되었으나, 이러한 인코딩 방법들은 실제로 구현하기는 복잡한 이론적인 방법이다. 따라서 본 논문에서는 실제로 구현 가능하면서도 자료구조의 크기를 $nH_0 + O(n)$비트로 줄이는 새로운 방법을 제안한다.

3. 압축 HSS 자료구조 구현 방법

3.1 기존의 압축 HSS 자료구조

압축하지 않은 HSS 자료구조는 어렵지 않게 구현할 수 있으나, 압축 HSS 자료구조는 직접 구현하지 않는다. 배열 R크기를 줄이는 인코딩 방법들을 구현하는 대신 비교를 위한 이론적인 크기만 계산한다. 즉 본 논문에서는 *rank/select* 연산 시간으로는 압축하지 않은 HSS 자료구조를 기준으로, 저장 공간으로는 압축한 HSS 자료구조를 기준으로 실제 구현 성능을 비교한다. R을 인코딩하게 되면 *rank/select* 연산을 위한 디코딩 과정이 필요하기 때문에 연산 시간이 더 걸린다. 또한 디코딩에 필요한 부가 정보에 대한 크기는 고려하지 않았기 때문에, 실제로 기존의 압축방식은 제시된 이론적 크기보다 더 커질 수 있다.

3.2 Huffman 코딩을 이용한 블록 크기 설정

본 논문은 문자의 빈도수에 따라 블록 크기를 다르게 설정하는 방법을 제안한다. 즉 빈도수가 잦은 문자의 오프셋은 짧은 비트를 차지하고 빈도수가 드문 문자의 오프셋은 긴 비트를 차지하도록 블록 크기를 잡는다. 이를 위해 첫 번째 방법은 문자 c의 오프셋 크기 l_c가 Huffman 코드 길이가 되도록 하여, 블록 크기를 2^{l_c}로 잡는 것이다. 이렇게 하면 B 크기는 변함없이 $2n+O(|A|)$ 비트를 유지하고 R 크기는 $n(H_0+1)$ 비트로 줄어든다.

$$|B| = \Sigma_c(n_c + n/2^{l_c}) = 2n + O(|A|)$$
$$|R| = \Sigma_c\, n_c l_c = n(H_0+1) \qquad (2)$$

예 3. Huffman 코딩을 적용하여 예 2의 문자열에 블록 크기를 각각 다르게 설정하기 위해, 문자별 빈도수를 이용해 아래의 Huffman 트리를 생성한다.

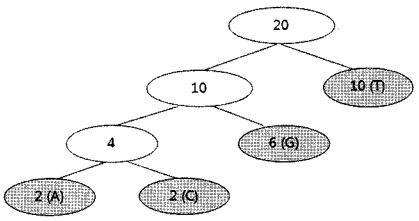


그림 2 문자열 빈도수에 따른 Huffman 트리 예제

Huffman 트리의 root로부터 각 문자별 빈도를 표시한 노드까지의 높이는 허프만 코드길이가 된다. (l_A = l_C = 3, l_G = 2, l_T = 1) 허프만 코드길이가 블록별 오프셋 길이로 사용되고, 블록크기의 지수로도 사용되어 아래와 같이 문자별로 융통성 있는 블록크기를 할당하게 된다.

A				A				
C								C
G		G	G	G			G	G
	T	T	T	T	T	TT	T	T	T

그림 3 Huffman 코드 길이에 따른 블록 크기 할당 예제

이들의 블록정보는 B배열에, 나머지 블록 내의 위치 정보는 R배열에 다음과 같이 저장된다.

B = 10101 10110 10100101010 1101010101010010101010
R = 000011 010001 010111001110 1001001001

3.3 Shannon 코딩을 이용한 블록 크기 설정

자료구조 크기가 B와 R을 모두 고려해서 최적의 크

기를 갖도록 블록 크기를 잡는 방법을 고려한다. 기존의 R을 인코딩하는 방법이나 Huffman 코드를 이용한 방법 모두 R의 크기만 nH_0비트로 줄이는 방법이다. 오프셋 크기를 Shannon 코딩[16] 길이 $l_c = \lfloor -\log(n_c/n) \rfloor$로 잡으면 정리 1을 만족해 $|B| + |R|$을 최소화 한다.

정리 1. 모든 문자 c에 대한 오프셋 길이 l_c가 다음을 만족하면 $|B| + |R|$을 최소화 한다.

$$-\log(n_c/n) - 1 < l_c < -\log(n_c/n)$$

증명. 최적의 오프셋 길이 집합 L에 대해, 임의의 문자 c에 대한 $l_c \in L$을 t비트 증가 혹은 감소시킨 L'은 L보다 $|B|+|R|$크기를 증가시킨다. t비트 증가시키는 경우 수식 (2)에 의해 $|B|$는 $n/2^{l_c} - n/2^{l_c+t}$ 만큼 감소, $|R|$은 tn_c만큼 증가한다. 증감분은 +가 되어야 하므로,

$n/2^{l_c}(1- 1/2^t) < tn_c$

$-\log(n_c/n) - \log t + \log(1- 1/2^t) < l_c$.

좌변은 t=1일 때 최대값 갖게 되므로,

$-\log(n_c/n) - 1 < l_c$.

t비트 감소시키는 경우에도 비슷한 방법으로 $l_c < -\log(n_c/n)$를 얻을 수 있다. □

예 4. Shannon 코딩을 적용하여 예 2의 문자열에 블록 크기를 각각 다르게 설정하고자 한다. 우선 문자별 빈도수를 이용해 Shannon coding 길이를 계산한다. ($l_A = l_C = 4$, $l_G = 2$, $l_T = 1$) 코드길이가 블록별 오프셋 길이로 사용되고, 블록크기의 지수로도 사용되어 아래와 같이 문자별로 융통성 있는 블록크기를 할당하게 된다.

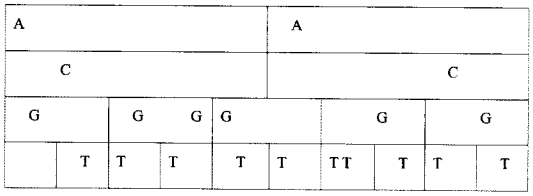


그림 4 Shannon 코딩에 따른 블록 크기 할당 예제

이들의 블록정보는 B배열에, 나머지 블록 내의 위치 정보는 R배열에 다음과 같이 저장된다.

B = 1010 1010 10100101010 11010101010100101010

R = 00000011 00100111 010111001110 1001001001

4. 실험 및 결과

본 장에서는 앞서 언급된 크게 3종류의 HSS 자료구조를 구현하였을 때 전체 자료구조의 크기와 *rank/select* 연산에 걸리는 시간을 비교하고자 한다. 실험 데이터로는 Pizza&Chili 사이트(http://pizzachili.dcc.uchile.cl)에서 제공하는 DNA($|A|$=16), protein($|A|$=27), XML문서($|A|$=97), 소스코드($|A|$=230), 영문문서($|A|$=239)의 5가지 실험셋을 사용했다. 이 실험셋은 문자열의 압축 및 인덱스 자료구조를 구현하는 연구분야에서 널리 사용되는 예제이다. 실험셋은 해당 데이터를 표현하는 직접적인 문자 외에 특수문자들도 포함하고 있다. 예를 들면 DNA의 경우 A, C, G, T, 네 종류의 기본 문자 외에도 미확인 부분 등을 나타내는 특수문자를 포함해 16종류 문자들로 이루어져 있다. 그리고 모든 실험은 펜티엄 듀얼 코어 2.33GHz CPU, 2GB 메모리, Windows XP환경에서 Microsoft Visual C++ 8.0로 구현된 코드를 사용하였다.

제안한 HSS 구현 방법이 배열 B에 대한 비트 *rank/select* 연산에 기반하기 때문에 사용한 자료구조를 명시한다. 비트 *rank/select*에 대한 기존의 효율적인 구현 방법들[17-19] 중 Gonzalez 등[17]이 제안한 알고리즘을 그대로 사용했다. 이 알고리즘의 실제 연산 시간과 크기는 다음과 같다. 연산 시간은 비트 1에 대한 *rank/select* 연산을 100만 번 수행했을 때 소요되는 시간의 합으로 측정한다. 비교하려는 50, 100, 200 MByte 문자열에 대해, 대략 100, 200, 400 Mbit 크기의 배열 B를 만들기 때문에 그에 대한 *rank/select* 연산 시간과 자료구조 크기가 표 2에 나와 있다.

표 2 비트 *rank/select* 자료구조의 연산 시간 및 크기

비트열 길이 (Mbit)	*rank* 시간 (ms)	*select* 시간 (ms)	자료구조 크기 (Mbit)
100	180	553	113
200	205	683	225
400	223	800	450

4.1 실험 1: 압축 크기 비교

50, 100, 200(MByte) 크기의 문자열에 대해 위에서 언급한 5종류의 실험셋을 입력으로 하여, 자료구조들의 크기를 비교 측정하였다. 그림 5에서 "HSS"는 압축하지 않은 HSS 자료구조, 이론적인 압축 HSS 자료구조인 "HSS-1"과 "HSS-2"는 각각 gap 인코딩으로 압축한 HSS 자료구조와 block identifier 인코딩으로 압축한 HSS 자료구조를 나타낸다. 본 논문에서 제안한 두 종류의 자료구조, 즉 Huffman 코딩과 Shannon 코딩을 이용하여 블록 크기를 조정한 HSS 자료구조는 각각 "Huf", "Sha"로 표기한다.

실험 결과 "Huf"와 "Sha" 모두 압축하지 않은 "HSS"에 비해 약 30%가 줄어드는 뛰어난 압축 효율을 보였다. 이 크기는 기존의 이론적인 HSS 압축 인코딩 중 가장 좋은 "HSS-2"에 근접하는 것이다. 각각의 특징을 살펴보면 다음과 같다. 이론적인 HSS 압축 인코딩인 "HSS-1"과 "HSS-2", 제안한 "Huf" 방식은 배열 B의 크기는 모두 같은 반면, 배열 R에 대해 제각각 다른 크기를 보여

준다. "Sha" 방식은 배열 B를 상대적으로 키우지만 배열 R를 줄여서 전체적으로 이득을 본다. 또한 "HSS-1"의 경우 거의 압축 효과를 확인할 수 없었고, "HSS-2"은 실험셋에 따라 성능이 떨어지는 경우를 볼 수 있었다.

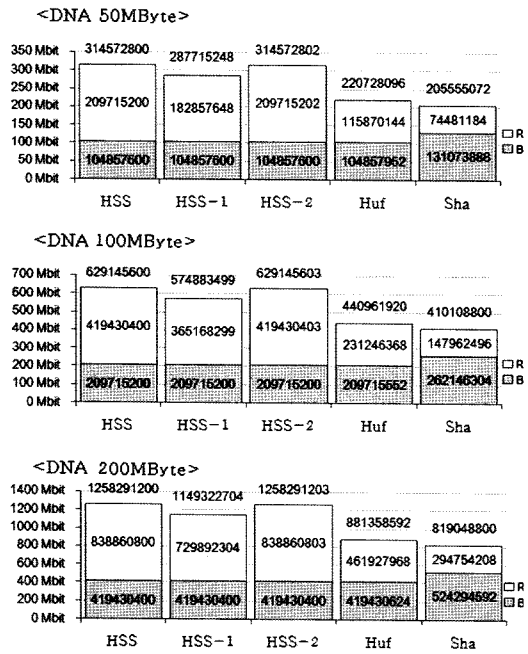


그림 5(a) DNA 문자열($|A|$=16)에 대한 *rank/select* 자료구조 크기

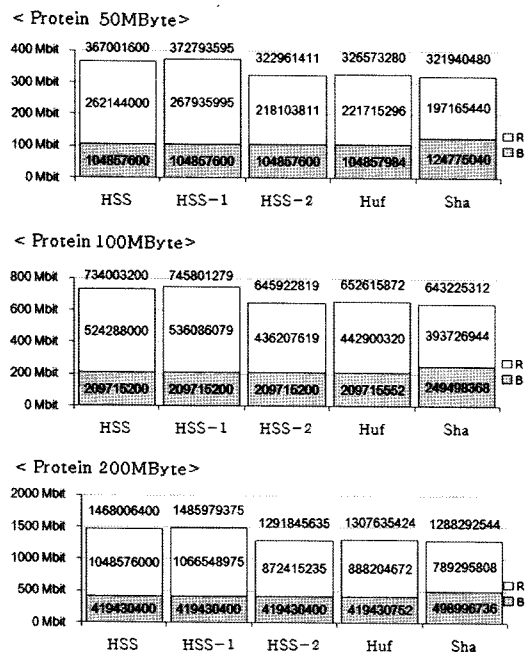


그림 5(b) Protein 문자열($|A|$=27)에 대한 *rank/select* 자료구조 크기

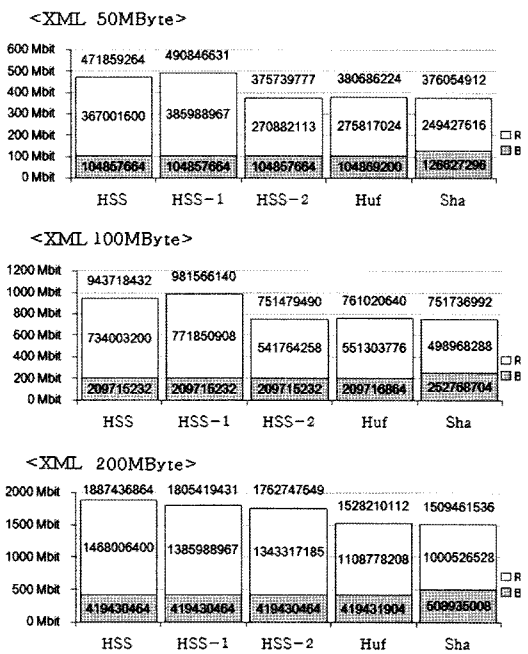


그림 5(c) XML 문자열($|A|$=97)에 대한 *rank/select* 자료구조 크기

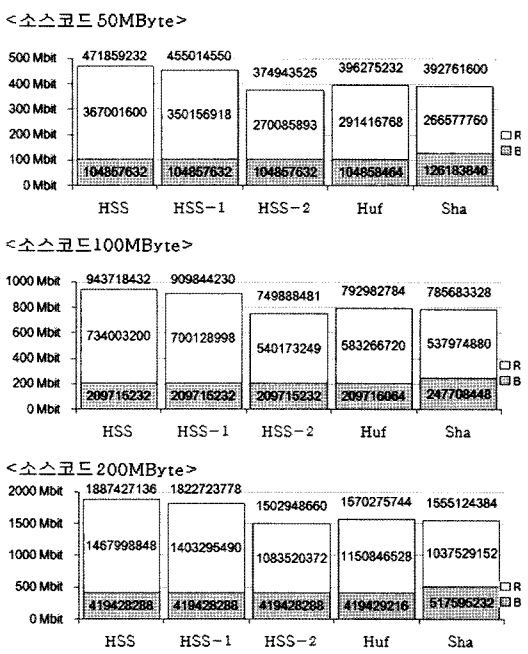


그림 5(d) 소스코드 문자열($|A|$=230)에 대한 *rank/select* 자료구조 크기

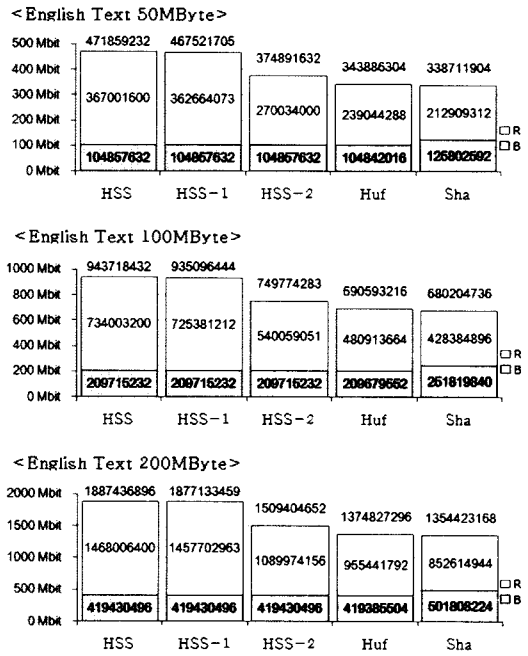
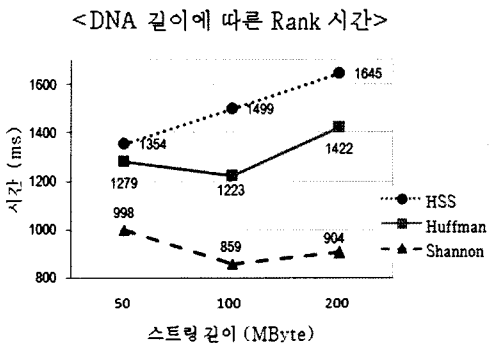


그림 5(e) 영문 문서(|A|=239)에 대한 rank/select 자료구조 크기

4.2 실험 2: *rank/select* 연산 시간 비교

실험 1과 마찬가지로 50, 100, 200(MByte) 크기 5종류의 실험셋을 입력으로 하여, *rank*와 *select* 연산에 걸리는 시간을 비교 측정하였다. 그림 6에서 "HSS"는 압축하지 않은 HSS 자료구조를 나타내고, "Huffman"과 "Shannon"은 각각 해당 코딩을 이용해 블록 크기를 조정한 자료구조들을 나타낸다. 이 때 제안한 방법은 문자 빈도수에 따라 블록 크기를 다르게 잡았기 때문에 *rank/select* 연산 시간이 문자별로 달라진다. 시간 측정을 위해 *rank/select* 연산은 모든 문자가 똑같은 빈도수로 일어난다고 보고 모든 문자에 대해 같은 비율로 *rank*와 *select* 연산을 각각 100만 번 수행한 시간의 합을 기록하였다.

실험 결과 문자열 길이와 상관없이 "Huffman", "Shannon", "HSS"의 순으로 빠른 *rank/select* 연산시간을 보여주었다. 따라서 본 논문이 제안한 구현 방식은 압축 HSS 자료구조의 공간 복잡도를 달성하면서도 압축하지 않은 HSS *rank/select* 연산시간을 얻는 구현 방식이라 하겠다. HSS 자료구조의 *rank* 연산은 배열 B에 대해 두 번의 비트 *select* 연산 호출을 수행하며, *select* 연산은 각각 한번씩 비트 *rank* 연산과 비트 *select* 연산을 호출한다. 따라서 표 2의 대략적인 비트 *rank/select* 시간을 제외해보면 제안한 HSS 자료구조가 사용하는 비트 *rank/select* 외 부가적인 연산 시간은 비트 *rank/select* 연산 시간에 못 미치는 것을 볼 수 있다.



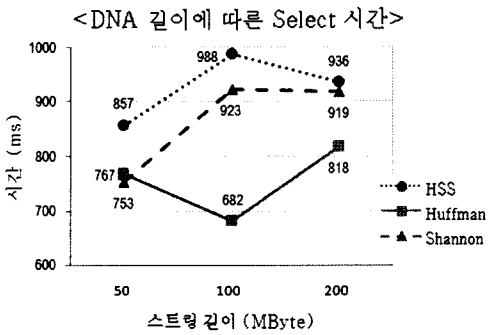
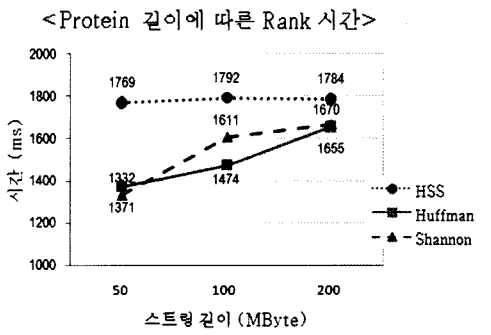


그림 6(a) DNA 문자열(|A|=16)에 대한 *rank/select* 연산 시간



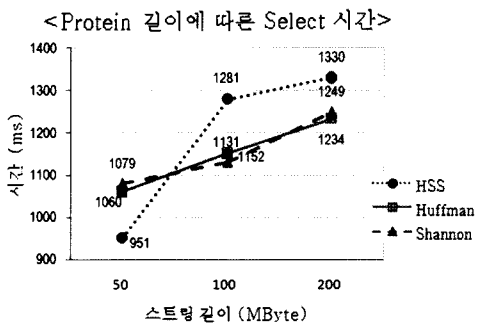
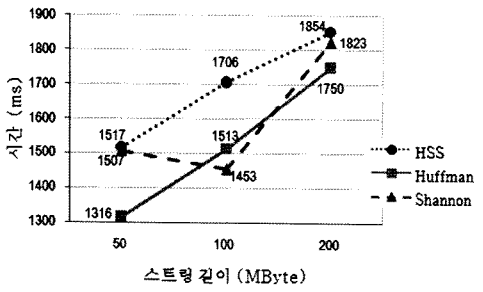


그림 6(b) Protein 문자열(|A|=27)에 대한 *rank/select* 연산 시간

<xml문서의 길이에 따른 Rank 시간>



<xml문서의 길이에 따른 Select 시간>

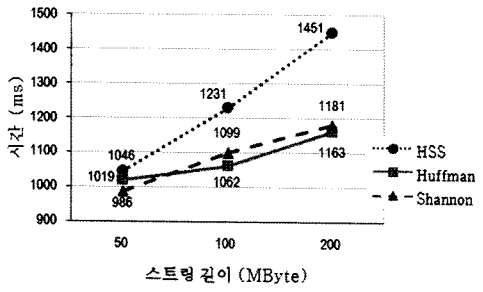
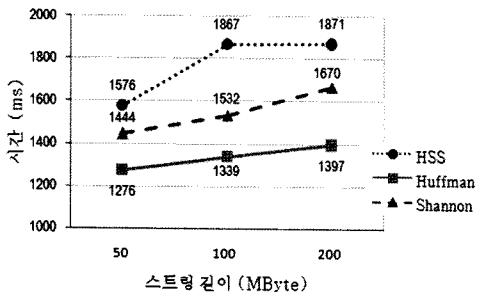


그림 6(c) XML 문자열(|*A*|=97)에 대한 *rank/select* 연산 시간

<소스코드의 길이에 따른 Rank 시간>



<소스코드의 길이에 따른 Select 시간>

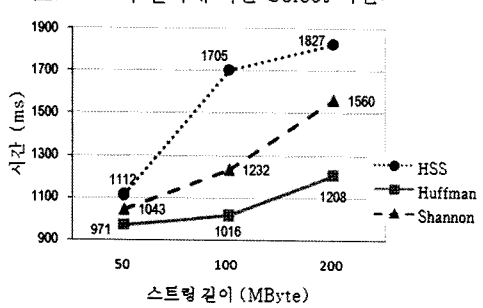
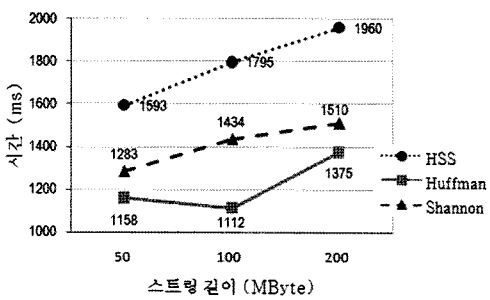


그림 6(d) 소스코드 문자열 (|*A*|=230)에 대한 *rank/select* 연산 시간

<영문문서의 길이에 따른 Rank 시간>



<영문문서의 길이에 따른 Select 시간>

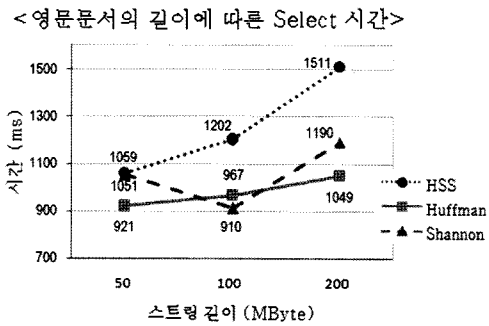


그림 6(e) 영문 문서(|*A*|=239)에 대한 *rank/select* 연산 시간

5. 결 론

본 논문에서는 문자의 빈도수를 고려해 압축 HSS *rank/select* 자료구조를 구현하는 효율적인 방법을 제시했다. 복잡한 인코딩을 하지 않으면서도 이론적인 $nH_0 + O(n)$ 비트 크기에 근접하고, 압축하지 않은 HSS 자료구조보다 빠른 *rank/select* 연산을 지원하는 자료구조를 얻었다.

참 고 문 헌

[1] E. M. McCreight, A space-economical suffix tree construction algorithm, *Journal of ACM*, 23, pp.262-272, 1979.

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

[3] R. Grossi and J. S. Vitter, Compressed suffix arrays and suffix trees with applications to text indexing and string matching, *SIAM Journal on Computing*, 35, pp.378-407, 2005.

[4] P. Ferragina and G. Manzini, Indexing compressed texts, *Journal of ACM*, 52, pp.552-581, 2005.

[5] 최용욱, 심정섭, 박근수, 접미사 배열을 이용한 시간과 공간 효율적인 검색, *한국정보과학회논문지*, 32, pp.260-267, 2005.

[6] G. Jacobson, Space-efficient static trees and graphs, In *Proceedings of FOCS*, 1989.
[7] D. R. Clark, *Compact Pat Trees*, PhD thesis, University of Waterloo, 1988.
[8] J. I. Munro and V. Raman, Succinct representation of balanced parentheses and static trees, *SIAM Journal on Computing*, 31, pp.762-776, 2001.
[9] R. Grossi, A. Gupta and J. S. Vitter, High-Order Entropy-Compressed Text Indexes. In *Proceedings of SODA*, 2003.
[10] W. Hon, K. Sadakane and W. Sung, Breaking a time-and-space barrier in constructing full-text indices. In *Proceedings of FOCS*, 2003.
[11] A. Golynski, J. I. Munro and S. S. Rao, Rank/ select operations on large alphabets: a tool for text indexing, *In Proceedings of SODA*, 2006.
[12] F. Claude and G. Navarro. Practical Rank/ Select Queries over Arbitrary Sequences. In *Proceedings of SPIRE*, 2008.
[13] G. Manzini, An analysis of the Burrows-Wheeler transform, *Journal of ACM*, 48, pp.407-430, 2001.
[14] K. Sadakane, New text indexing functionalities of the compressed suffix arrays, *Journal of Algorithms*, 48, pp.294-313, 2003.
[15] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets, In *Proceedings of SODA*, 2002.
[16] T. Cover and J. Thomas. *Elements of Information Theory*, Wiley-Interscience, 1991.
[17] R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical Implementation of Rank and Select Queries. In *Poster Proceedings of 4th WEA, 2005.*
[18] D. Okanohara and K. Sadakane, Practical entropy-compressed rank/select dictionary, In *Proceedings of ALENEX*, 2007.
[19] 박치성, 조준하, 김동규, Succinct 표현의 효율적인 구현을 통한 압축된 써픽스 배열 생성. *한국정보과학회 제 32회 추계학술발표회 2005.*

권 유 진
2002년~2007년 고려대학교 정보통신대학 컴퓨터학과 학사. 2007년~2009년 서울대학교 전기,컴퓨터공학부 석사. 2009년~현재 IBM Korea 연구소 Ubiquitous Computing Lab. 관심분야는 컴퓨터 이론, 알고리즘, 생물정보학, 의료정보 시스템

이 선 호
2002년 서울대학교 컴퓨터공학부 학사 2002년~2009년 서울대학교 전기,컴퓨터공학부 박사. 2009년~현재 한양대학교 BK21 postdoc. 관심분야는 컴퓨터 이론, 알고리즘, 생물정보학

박 근 수
정보과학회논문지 : 시스템 및 이론 제 36 권 제 2 호 참조