Succinct 표현을 위한 *rank* 함수의 효율적인 구현

김지은°, 김동규
부산대학교 컴퓨터공학과
{jekim°, dkkim}@islab.ce.pusan.ac.kr

Efficient Implementation of the *rank* Function for Succinct Representation

Ji Eun Kim°, Dong Kyue Kim
Dept. of Computer Engineering
Pusan National University, Busan 609-735, Korea

요약

컴퓨터상에서 집합, 트리, 그래프 등의 이산 객체(discrete object)는 메모리 주소와 같은 정수 값으로 표현되어 왔다. Succinct 표현(succinct representation)은 이산 객체를 이진화 하여 표현하는 방법으로, 정수 값으로 표현된 이산객체보다 적은 공간에 저장된다. Succinct 표현의 데이터는 이산 객체의 개별적 저장 위치를 파악하기 위해서 *rank*와 *select* 함수가 기본적으로 필요하다. *rank* 와 *select* 함수를 MBRAM 모델 상에서 $O(1)$시간에 계산하는 알고리즘이 제시되었다. MBRAM 모델은 비트연산을 unit cost로 수행하는 모델이다. 그러나 범용 컴퓨터는 바이트 단위의 기본 명령어를 제공하므로 비트연산을 unit cost에 수행할 수 없다. 본 논문은 범용 컴퓨터에서 효율적으로 *rank* 함수를 구현하는 방법을 제시한다. *rank* 함수의 효율적인 구현은 FM-index 등의 패턴 검색에 유용하다.

1. 서론

컴퓨터상에서 집합, 트리, 그래프 등의 이산 객체는 메모리 주소와 같은 정수 값으로 표현되어 왔다. Succinct 표현은 이러한 이산 객체를 이진화 하여 표현하는 방법이다. 이산 객체를 이진화 하여 표현한 것은 정수 값으로 표현한 것에 비해 적은 저장 공간에 저장되므로 크기가 큰 데이터를 다루는데 유용하다. Succinct 표현은 이산 객체뿐만 아니라 함수, 순열 등 다양한 대상에 적용되어 연구되고 있다[5-7, 8, 9, 12, 13, 14].

Succinct 표현을 다루는데 필요한 기본 함수는 *rank* 와 *select* 함수[1-4, 7]이며, 다음과 같이 정의한다. $rank(i)$는 비트 스트링 S에서, i보다 작거나 같은 위치에 있는 '1'의 개수를 반환한다. $select(j)$는 비트 스트링 S에서, j 번째 발생하는 '1'의 위치를 반환한다. 본 논문은 *rank* 와 *select* 함수 중 *rank* 함수의 알고리즘들을 기술하고, *rank* 함수의 효율적인 구현 방법을 제시한다. *rank* 함수의 효율적인 구현은 succinct 표현의 이산 객체 구현에 용이할 뿐 아니라 FM-index와 같은 패턴 검색에 유용하다[8, 9, 12, 13].

rank 함수는 1988년에 Jacobson에 의해 처음으로 제안된 함수이다[3, 4]. Jacobson은 2-level directory라는 자료 구조에 일부 *rank* 값을 저장한다. 입력되는 비트 스트링의 길이가 n일 때, *rank* directory를 생성하는 시간은 $O(n)$, *rank* 함수의 수행시간은 $O(\log n)$, 공간복잡도는 $o(n)$비트이다.

Clark은 1996년에 Jacobson의 알고리즘을 향상시켜, MBRAM 모델[11] 상에서 *rank* 함수를 $O(1)$시간에 수행하는 방법을 제안하였다[1, 2]. Clark은 Jacobson의 2-level directory와 하나의 테이블을 사용하며, 입력되는 비트 스트링의 길이가 n일 때, 자료 구조의 생성 시간은 $O(n)$, 공간복잡도는 $o(n)$비트이다.

Munro는 2001년에 데이터의 succinct 표현을 위한 3-level directory의 자료 구조를 제안하였는데, 이것은 이전의 방법들보다 더 논리적이다[7]. Munro의 방법은 MBRAM 모델 상에서, 입력되는 비트 스트링의 길이가 n일 때, 자료 구조의 생성 시간이 $O(n)$, *rank* 함수의 수행시간이 $O(1)$, 공간 복잡도가 $o(n)$비트이다.

Clark과 Munro의 방법은 *rank* 함수를 $O(1)$시간에 수행하게 하는 강력한 자료구조이다. 그러나 Clark과 Munro는 이론적 계산 모델인 MBRAM 모델을 가정하기 때문에 범용 컴퓨터에 구현 시 두 가지 문제가 있다. 첫 번째 문제점은 MBRAM 모델이 비트연산을 unit cost에 수행하는 반면, 범용 컴퓨터는 비트연산을 지원하지 않는다는 점이다. 범용 컴퓨터는 바이트 단위의 기본 명령어를 지원하기 때문에 바이트 단위의 연산만을 unit cost에 수행한다. 두 번째 문제점으로는 MBRAM 모델은 *word*길이가 고정되지 않지만 범용 컴퓨터의 *word*길이는 w로 고정되어 있어서 2^w 바이트이내의 길이만 다룰 수 있다.

본 논문은 길이가 2^w바이트 이내의 스트링만 입력된다는 가정 하에서 바이트 단위의 연산만을 사용하는 *rank* directory구조를 제안한다. 본 논문에서 제안하는 방법의 타당성은 이론

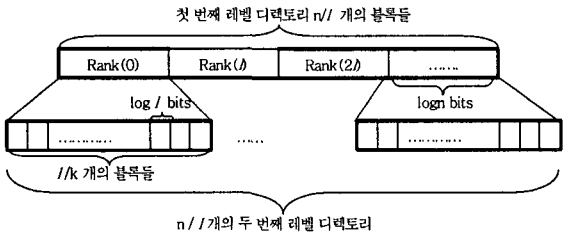
적으로 증명되며, 실험을 통해서 뒷받침된다.

본 논문의 2장은 기존의 알고리즘들을 기술한다. 3장은 본 논문의 새로운 아이디어를 제시하며, 4장에서 기존의 알고리즘들과 논문의 방법을 구현한 결과를 비교 분석한다.

2. 기존 알고리즘

2-1. Jacobson의 방법

Jacobson은 2-level directory의 자료 구조에 일부의 *rank* 값을 저장한다. 입력 스트링의 길이가 n 비트일 때, 1st-level은 입력 스트링을 길이가 $\log^2 n$ 비트인 블록들로 분할한다. 각 블록의 *rank* 값을 1st-level directory에 저장한다. 2nd-level은 길이가 $\log^2 n$ 비트인 각 블록들을 길이가 $\log n$ 비트인 블록들로 다시 분할한다. 하나의 블록은 $\log n$ 개의 블록으로 분할된다. 그 후, 2nd-level directory에 나누어진 블록들의 *rank* 값을 저장한다. 1st-level의 블록 크기를 $l = \log^2 n$ 비트, 2nd-level의 블록 크기를 $k = \log n$ 비트로 하여 Jacobson의 *rank* directory는 [그림1]과 같이 표현된다.



[그림 1] Jacobson의 *rank* 디렉토리

$rank(i)$는 나눗셈 연산에 의해 계산된다. 먼저, 1st-level directory에서 i를 넘지 않고, i에 가장 가까운 블록은 $b_1 = \frac{i}{l}$ 이다. 2nd-level directory에서 i를 넘지 않고, i에 가장 가까운 블록은 $b_2 = \frac{(i \bmod l)}{k}$ 이다. 1st-level directory의 $rank(l \times b_1)$ 값과, 2nd-level directory의 $rank(k \times b_2)$ 값의 합은 $rank(b_1 \times l + b_2 \times k)$ 의 값이다. 남은 비트 $(b_1 \times l + b_2 \times k + 1).....i$는 최대 k 비트를 넘지 않으므로 직접 입력 스트링을 읽으면서 1의 수를 센다. 마지막 구해진 1의 수와 $rank(b_1 \times l + b_2 \times k)$ 값의 합이 $rank(i)$의 값이다.

$rank(i)$는 $\log n + 2\log\log n + \log n = O(\log n)$ 시간에 수행되며, *rank* directory는 $\frac{n + 2n\log\log n}{\log n} = o(n)$ 비트에 저장된다.

2-2. Clark의 방법

Clark은 Jacobson의 알고리즘을 향상 시켜, MBRAM 모델 상에서 $O(1)$의 시간복잡도로 *rank*를 계산한다. Clark은 Jacobson의 *rank* directory와, 하나의 테이블을 사용한다. 테이블은 $\frac{\log n}{c}$ $(c \geq 2)$비트로 나타낼 수 있는 각 비트맵의 *rank* 값을 저장한다. 테이블은 $c=2$일 때, $\sqrt{n}$비트이므로, 총 사용되는 공간은 $o(n)$ 비트이다.

2-3. Munro의 방법

Munro는 succinct 표현을 위해 3-level directory 구조를 제안하였다. 1st-level의 블록 길이는 $b = \log^2 n$ 비트, 2nd-level의 블록 길이는 $\log^2 b = (2\log\log n)^2$ 비트, 3rd-level의 블록 길이는 $2\log\log n$ 비트이다. 테이블은 $\log\log n$ 비트에 대해서 만들어진다. *rank* directory와 테이블의 저장 공간은

$$\frac{n}{\log n} + \frac{n}{2\log\log n} + \frac{n\log(2\log\log n)}{2\log\log n} + \log n \cdot \log\log\log n = o(n)$$

비트이고, *rank* 함수는 MBRAM 모델 상에서 $O(1)$의 시간에 수행된다.

3. *rank* 함수의 새로운 구현

본 장에서는 범용 컴퓨터에 가장 적합하도록 *rank*함수를 구현하는 방법을 제안한다. 범용 컴퓨터가 제공하는 기본 명령어는 바이트 단위이므로 바이트 단위의 연산만을 unit cost로 수행한다. 본 논문에서 제안하는 방법은 바이트 단위의 연산만을 사용하는 *rank*함수를 구현하는 것이다.

rank directory 구조를 결정하는 두 가지 요소는 블록의 길이와 각 블록을 저장하는 공간의 길이이다. 이 두 요소는 입력되는 비트 스트링의 길이에 의해서 결정된다. 입력되는 비트 스트링의 길이가 $N = 2^{256}$비트라고 가정하고, Munro의 3-level *rank* directory를 구축한 것이 [표1]이다.

[표1] $N = 2^{256}$비트인 비트 스트링의 *rank* directory

	블록 길이		블록 저장 공간의 길이	
입력 길이	n	$N = 2^{256}$	n	$N = 2^{256}$
1st-level	$\log^2 n$	64k	$\log n$	256(32)
2nd-level	$(2\log\log n)^2$	256	$2\log\log n$	16
3rd-level	$2\log\log n$	16	$2\log(2\log\log n)$	8
테이블	$\log\log n$	8	$\log(2\log\log n)$	4 (8)

[표1]은 테이블의 저장 공간의 길이를 제외하고는 모든 길이가 8의 배수가 됨을 보여준다. 테이블의 저장 공간의 길이를 8로 변경하면 모든 길이가 바이트 단위가 된다. 테이블은 크기가 매우 작기 때문에 저장 공간을 8로 하더라도 전체 공간에 큰 영향을 주지 않는다. [표1]이 나타내는 *rank* directory는 바이트 단위의 연산만 사용하도록 *rank* 함수를 지원한다. 본 논문의 방법은 입력되는 비트 스트링의 길이에 상관없이 [표1]의 *rank* directory구조를 동일하게 적용하는 것이다. 일반적으로 입력되는 비트 스트링의 길이는 2^{32}비트를 넘지 않으므로 1st-level의 블록 저장 공간의 길이를 32비트로 결정할 수 있다.

본 논문의 방법은 네 가지 면에서 유용하다. 첫째, 입력되는 비트 스트링의 길이에 관계없이 *rank* directory의 구조가 바이트 단위로 고정되기 때문에 구현이 용이하다. 둘째, 바이트 단위의 연산만으로 *rank* directory를 생성하므로 생성 시간이 빨라진다. 셋째, 바이트 단위의 연산만으로 *rank* 함수를 수행하므로 수행 속도가 빠르다. 넷째, 본 논문의 방법이 Munro의 방법보다 적은 공간을 사용한다. 임의의 입력 길이 n에 대하여, Munro의 방법과 비교한 것이 아래와 같으므로 본 논문이 공간상으로 Munro의 방법보다 효율적이다.

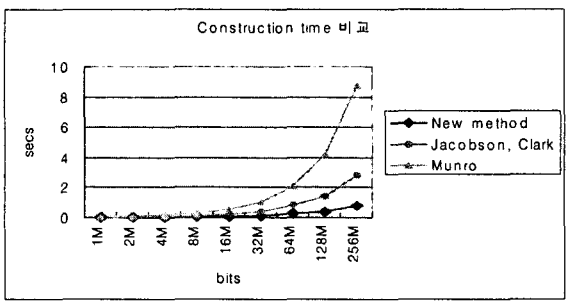
$$\frac{n}{256}+\frac{n+4n}{16}+256\cdot 8 <$$

$$\frac{n}{\log n}+\frac{n+n\log(2\log\log n)}{2\log\log n}+\log n\cdot \log\log\log n$$

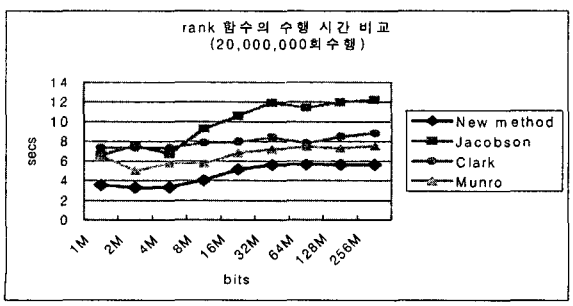
4. 실험 및 결론

본 장은 네 개의 *rank* directory를 구현하여 그 성능을 비교 분석한다. Jacobson의 방법, Clark의 방법, Munro의 방법, 그리고, 본 논문에서 제시하는 방법으로 구현하였다.

[그림2]는 *rank* directory의 생성 시간을 비교한 그래프이다. 본 논문의 방법이 가장 빠르며, 그 다음이 Jacobson과 Clark, 가장 느린 것이 Munro의 방법이다. **[그림3]**은 *rank* 함수의 수행시간을 비교한 그래프이다. *rank* 함수의 수행시간은 측정할 수 없기 때문에 20,000,000회 수행하였다. 본 논문의 방법이 가장 빠르며, Munro, Clark, Jacobson의 방법 순으로 느려진다. **[그림2]**와 **[그림3]**을 통해 본 논문의 방법이 생성 시간과 *rank* 함수의 수행시간이 가장 효율적임을 알 수 있다.

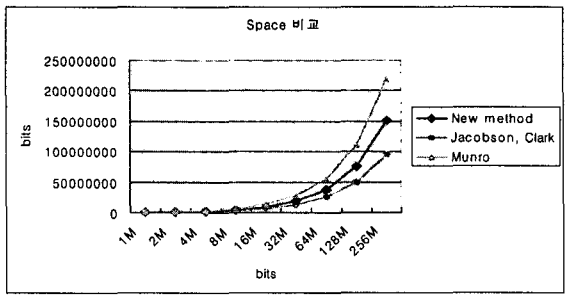


[그림2] *rank* directory의 생성시간 비교



[그림3] *rank* 함수의 수행 시간 비교

[그림4]는 *rank* directory를 생성하는데 사용되는 공간을 비교한 그래프이다. 가장 많은 공간을 차지하는 것은 Munro의 방법이고, 두 번째는 본 논문의 방법이고, 가장 작은 공간을 차지하는 것이 Jacobson과 Clark의 방법이다. 본 논문의 방법이 공간상으로 가장 효율적이지는 않지만 $o(n)$ 비트의 공간을 차지하고, Munro의 방법보다 효율적이다.



[그림4] *rank* directory의 공간 비교

5. 참고 문헌

[1] D. R. Clark, Compact pat trees, Ph.D. thesis, University of Waterloo, Canada, 1996.

[2] D. R. Clark and J. I. Munro, Efficient suffix trees on secondary storage, In Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms (SODA), 1996, 538-544

[3] G. Jacobson, Succinct static data structure, Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, 1988.

[4] G. Jacobson, Space-efficient static trees and graphs, In Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS), 1989, 549-554.

[5] J. I. Munro, R. Raman, V. Raman, and S. S. Rao, Succinct representation of permutations, Proceedings of ICALP 2003, LNCS 2719. 345-356

[6] J. I. Munro and S. S. Rao, Succinct representations of functions, Proceedings of ICALP 2004, LNCS 3142. 1006-1015

[7] J. I. Munro and V. Raman., Succinct representation of balanced parentheses, static trees and planar graphs, SIAM J. Computing 31 2001, 762-776.

[8] K. Sadakane, Compressed text databases with efficient query algorithm based on the compressed suffix array, Int. Symp. Algorithms and Computation, 2000, 410-421

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

[10] P. Ferragina and G. Manzini Opportunistic data structures with applications, In Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS). 2000, 390-398

[11] P. van Emde Boas. Machine models and simulations. In J. van Leeuwen, editor, Handbook of Theoretical Computer Science, Volume A, Algorithms and Complexity, 1-66, Elsevier, Amsterdam, New York, Oxford, Tokyo, 1990

[12] R. Grossi and J. S. Vitter, Compressed suffix arrays and suffix trees with applications to text indexing and string matching, ACM Symp. Theory of Computing, 2000, 397-406,

[13] W. K. Hon and K. Sadakane. Space economical algorithms for finding maximal unique matches, Symp. Combinatorial Pattern Matching. 2002, 144-152

[14] X. He, M.-Y. Kao, and H.-I. Lu, Linear-time succinct encodings of planar graphs via canonical orderings, SIAM J. Discrete Math., 12, 1999, 317-325