2차원 배열의 Succinct 표현을 위한 Rank 및 Select 함수

박치성[1,O] 김민환[2] 김동규[3]
[1,2] 부산대학교 컴퓨터공학과
[3] 한양대학교 전자통신컴퓨터공학부
[1] cspark@islab.ce.pusan.ac.kr
[2] mhkim@pusan.ac.kr
[3] dqkim@hanyang.ac.kr

Rank and Select Functions for Succinct Representation of Two-Dimensional Arrays

Chi Seong Park[1,O] Minhwan Kim[2] Dong Kyue Kim[3]
[1,2] Dept. of Computer Science & Engineering, Pusan National University, Busan 609-735, Korea
[3] Dept. of Electronics and Computer Engineering, Hanyang University, Seoul 133-791, Korea

요 약

집합이나 배열의 원소, 트리의 노드, 그래프의 정점과 간선 등과 같은 이산 객체는 일반적으로 주기억장치의 논리적 주소 값과 같은 정수로 표현되어 왔다. Succinct 표현은 이와 같은 n개의 이산 객체를 $O(n)$ 비트에 표현하는 방법이다. 대부분의 succinct 표현은 *rank*와 *select*라는 함수를 기본적으로 사용하며, 다양한 연구들에 의해 현재 *rank*와 *select* 함수는 $o(n)$ 비트만을 사용하여 $O(1)$ 시간에 수행될 뿐만 아니라, 실제로도 실용적으로 구현되었다. 본 논문에서는 $n \times n$ 배열, 즉 2차원 비트 스트링에 대한 *Rank* 및 *Select* 함수를 새롭게 정의한다. 또한, $O(n^2 \log n)$ 비트를 사용하여 $O(1)$ 시간에 *Rank* 질의를 수행하고 $O(\log n)$ 시간에 *Select* 질의를 수행하는 방법과, 보다 적은 $O(n^2)$ 비트를 사용하면서 $O(\log n)$ 시간에 *Rank* 질의를 수행하고 $O(\log^2 n)$ 시간에 *Select* 질의를 수행하는 방법을 제안한다. 본 논문에서 정의하는 2차원 배열 상의 *Rank*와 *Select* 함수는 이미 개발된 2차원 상의 써픽스 트리 등을 기반으로 향후 개발될 2차원 상의 압축된 인덱스 자료구조나 이미지 프로세싱 등에 유용하게 사용된다.

1. 서 론

집합(set)이나 배열(array)의 원소(element), 트리(tree)의 노드(node), 그래프(graph)의 정점(vertex)과 간선(edge) 등과 같은 이산 객체(discrete object)들은 일반적으로 주기억장치의 논리적 주소(logical address) 값과 같은 정수(integer)로 표현되어 왔다. 이러한 방법으로 n개의 이산 객체를 표현(representation)할 경우 필요한 저장공간은 $O(n)$ 워드(word), 다시 말해 $O(n \log n)$ 비트(bit)이다.

Succinct 표현은 n개의 이산 객체를 $O(n)$ 비트에 표현하는 방법으로써, 자료구조를 succinct 표현하면 적은 공간의 사용이 가능하다. 이러한 장점으로 인해 집합이나 트리 [1], 그래프 [2, 3], static dictionary 또는 dynamic dictionary [4, 5]는 자료구조뿐만 아니라 순열(permutation) [6], 함수(function) [7] 등도 1차원의 비트 스트링(bit string)으로 succinct 표현하는 기법들이 다양하게 연구되어 왔다. 또한, 스트링 처리(string processing) 분야에서 사용되는 압축된 써픽스 트리(compressed suffix tree)나 압축된 써픽스 배열(compressed suffix array) [8, 9, 10, 11, 12, 13], FM-index [14, 15] 등의 전체 텍스트 인덱스 자료구조(full-text index data structure)나, 생물정보학(bioinformatics) 분야에서 다루는 방대한 양의 DNA 시퀀스(sequence) 자료들을 효율적으로 저장하기 위해서도 succinct 표현이 사용되고 있다.

대부분의 succinct 표현은 *rank*와 *select*라는 함수를 기본적으로 사용한다. 정적인(static) 비트 스트링 $A[0..n-1]$에 대하여, *rank*와 *select* 함수는 다음과 같이 정의된다.

- $rank(x) = y$
  y : $A[0..x]$에 포함된 1의 수
- $select(y) = x$
  x : A의 처음부터 y번째 1의 위치

예를 들어, $A = 10010110$이 주어졌을 때 $rank(4) = 2$, $select(2) = 3$이다. 이 정의에 따르면 *rank*와 *select* 함수가 역함수 관계를 형성하고 있음을 알 수 있다.

*rank*와 *select* 함수의 수행은 이산 객체의 접근을 의미하기 때문에, *rank*와 *select* 함수의 성능이 succinct 표현된 자료구조의 성능을 결정하게 된다. *rank*와 *select* 함수를 처음 제안한 Jacobson [3, 4]은 $O(\log n)$ 시간에 질의가 수행되는 $o(n)$ 비트의 directory 기반 자료구조를 개발하였으며, 이후 Clark [16], Munro and Raman [1], Miltersen [17], Kim et al. [18] 등의 연구들을 통해, 현재 *rank*와 *select* 함수는 $o(n)$ 비트를 사용하여 $O(1)$ 시간에 수행 가능하며, 실용적인 구현도 가능하다.

본 논문에서는 $n \times n$ 배열, 즉 2차원 비트 스트링에 대한 *Rank* 및 *Select* 함수를 새롭게 정의한다. 또한, $O(n^2 \log n)$ 비트를 사용하여 $O(1)$ 시간에 *Rank* 질의를 수행하고 $O(\log n)$ 시간에 *Select* 질의를 수행하는 방법과, 보다 적은 $O(n^2)$ 비트를 사용하면서 $O(\log n)$ 시간에 *Rank* 질의를 수행하고 $O(\log^2 n)$ 시간에 *Select* 질의를 수행하는 방법을 제안한다. 본 논문에서 정의하는 2차원 배열 상의 *Rank*와 *Select* 함수는 이미 개발된 2차원 상의 써픽스 트리 [19] 등을 기반으로 향후 개발될 2차원 상의 압축된 인덱스 자료구조나 이미지 처리(image processing) 등에 유용하게 사용될 수 있다.

본 논문은 총 5장으로 구성된다. 먼저 2장에서 2차원 배열 상의 *Rank* 및 *Select* 함수를 정의한 후, 3장에서 *Rank* 함수를 $O(1)$ 시간에 수행하고, 이를 이용하여 *Select* 함수를 $O(\log n)$ 시간에 수행할 수 있는 간단한 방법을 제안한다. 4장에서는 3장의 방법보다 느리지만 보다 효율적으로 공간을 사용하는 방법을 제안하고, 5장에서 결론을 맺는다.

2. 문제 정의

$n \times n$ 크기의 정적인 비트 스트링 $A[0..n-1, 0..n-1]$이 주어졌다고 가정하자. A 상의 $i(0 \le i \le i' \le n-1)$번째 row와 $j(0 \le j \le j' \le n-1)$번째 column의 교점 상에 있는 비트 위치를 $p(i,j)$라 하자. 또한, 두 점 $p(i,j)$와 $p(i',j')$에 의해 만들어지는 A 상의 직사각형을 $A[p(i,j):p(i',j')]$라 하자. (그림 1)

임의의 $p(i,j)$에 대하여, 우리는 A의 *Rank* 및 *Select* 함수를 다음과 같이 정의한다. (그림 2)

- $Rank_p(x) = y$
  y : $A[p(i,j):p(i+x,j+x)]$ 내에 포함된 1의 수
- $Select_p(y) = x$
  x : $A[p(i,j):p(i+x,j+x)]$ 내에 포함된 1의 수를 z라 했을 때, $y \le z$를 만족하는 x들 중 최소의 값

이 정의에 따르면 1차원에서의 *rank*, *select* 함수와 마찬가지로 2차원의 *Rank*, *Select* 함수 역시 역함수 관계를 형성하고 있다.

질의시마다 매번 A 전체를 스캔(scan)하는 원시적인(naive) 방법을 사용할 경우, 별도의 자료구조를 구축하지 않아도 되지만 $Rank_p(x)$와 $Select_p(y)$ 질의에 $O(n^2)$ 시간이 필요하게 된다. 반면 A의 모든 $p(i,j)$에 대한 $Rank_p(x)$ 값을 다 저장하는 자료구조를 구축할 경우, $Rank_p(x)$ 질의는 $O(1)$ 시간에 수행되고 $Select_p(y)$ 질의는 $Rank_p(x)$ 값들을 이진 검색(binary searching)하여 $O(\log n)$ 시간에 수행될 수 있지만, 이 자료구조를 저장하기 위해 $O(n^3 \log n)$ 비트의 공간을 사용하게 된다.

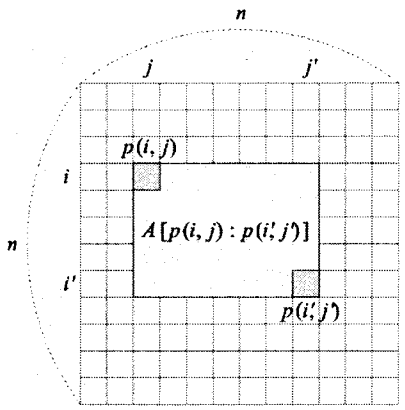

그림 1. 직사각형 $A[p(i,j):p(i',j')]$

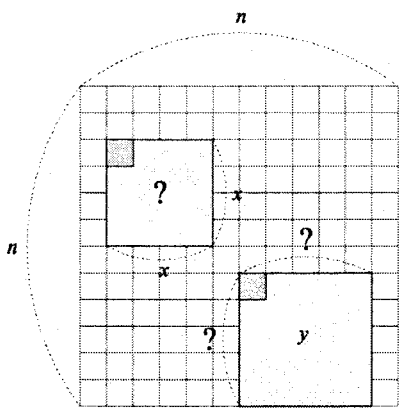

그림 2. $Rank_p(x)$ 및 $Select_p(y)$

본 논문의 3장과 4장에서 다루려는 문제는 되도록 적은 공간을 사용하면서도 가능한 빠른 시간에 $Rank_p(x)$와 $Select_p(y)$ 질의를 수행 가능하게 하는 자료구조를 구축하는 것이다.

3. 간단한 해결 방법

본 장에서는 $O(n^2 \log n)$ 비트를 사용하면서 $O(1)$ 시간에 $Rank_p(x)$ 질의를, $O(\log n)$ 시간에 $Select_p(y)$ 질의를 수행 가능하게 하는 자료구조를 소개한다. 이 자료구조는 A의 2차원적인 특성을 이용하여 매우 간단하다.

먼저 임의의 $p(i,j)$에 대한 $Rect_I(i,j)$, $Rect_{II}(i,j)$, $Rect_{III}(i,j)$, $Rect_{IV}(i,j)$를 각각 다음과 같이 정의한다. (그림 3)

- $Rect_I(i,j) = A[p(0,0):p(i,j)]$
- $Rect_{II}(i,j) = A[p(0,j):p(i,n-1)]$
- $Rect_{III}(i,j) = A[p(i,0):p(n-1,j)]$
- $Rect_{IV}(i,j) = A[p(i,j):p(n-1,n-1)]$

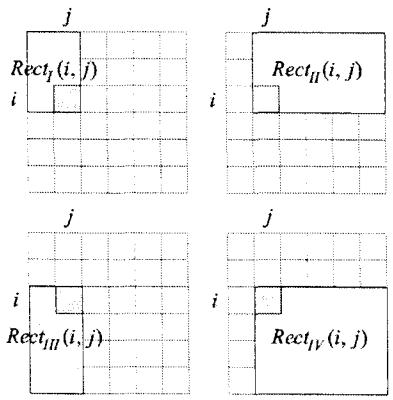

그림 3. $p(i,j)$에 대한 $Rect_I(i,j)$, $Rect_{II}(i,j)$, $Rect_{III}(i,j)$, $Rect_{IV}(i,j)$

$Rect_I(i,j)$, $Rect_{II}(i,j)$, $Rect_{III}(i,j)$, $Rect_{IV}(i,j)$는 A 상에서 각각 $p(i,j)$의 좌 상단, 좌 하단, 우 상단, 우 하단에 위치하게 된다.

이제 자료구조 $RECT_I(0..n-1, 0..n-1)$을 구축한다. 이 때, $RECT_I[i,j]$에는 $Rect_I(i,j)$ 내에 포함된 1의 수를 기록하며, A 전체를 한 번 스캔하여 $RECT_I$을 구축할 수 있다.

$RECT_{II}$, $RECT_{III}$, $RECT_{IV}$도 $RECT_I$과 동일한 방법으로 함께 구축할 수 있으므로, $RECT_I$, $RECT_{II}$, $RECT_{III}$, $RECT_{IV}$의 전체 구축 시간은 $O(n^2)$이다. 각각 n^2개의 엔트리(entry)를 필요로 하고, 하나의 엔트리가 $O(\log n^2)$ 비트를 사용하기 때문에 전체 자료구조가 사용하는 공간은 $O(n^2 \log n)$ 비트이다.

- $RECT_I$, $RECT_{II}$, $RECT_{III}$, $RECT_{IV}$ 자료구조는 $O(n^2)$ 시간에 구축되며, $O(n^2 \log n)$ 비트를 사용한다. 또한, 임의의 $Rect_I(i,j)$, $Rect_{II}(i,j)$, $Rect_{III}(i,j)$, $Rect_{IV}(i,j)$의 1의 수를 $O(1)$ 시간에 반환한다. (1)

(1)을 기반으로 하여 다음과 같이 $Rank_p(x)$ 값을 구할 수 있다. (그림 4)

- $Rank_p(x) = RECT_I(n-1, n-1)$
  $- RECT_I(i-1, j+x-1)$
  $- RECT_{II}(i+x-1, j+x)$
  $- RECT_{III}(i, j-1)$
  $- RECT_{IV}(i+x, j)$ (2)

따라서 $Rank_p(x)$ 질의는 $O(1)$ 시간에 수행된다.

$Select_p(y) = x$ 질의의 경우에는, y와 $Rank_p(x)$ 질의를 비교하면서 n 상에서 x를 이진 검색하는 방법을 사용한다. 비교 연산에 필요한 $Rank_p(x)$ 질의가 $O(1)$ 시간에 수행되고 이진 검색이 이루어지는 구간의 최대 길이가 n이므로, $Select_p(y)$ 질의는 $O(\log n)$ 시간에 수행된다.

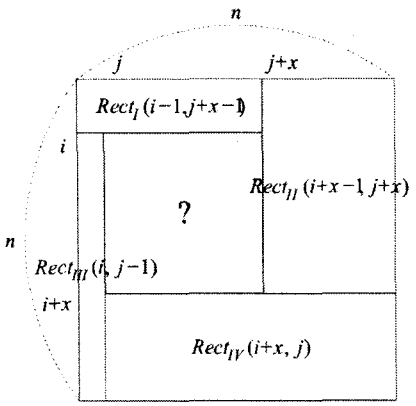

그림 4. $Rank_p(x)$를 구하는 간단한 방법

4. 공간 효율적인 방법

본 장에서는 A와 동일한 크기인 $O(n^2)$ 비트를 사용하여 $O(\log n)$ 시간에 $Rank_p(x)$ 질의를, $O(\log^2 n)$ 시간에 $Select_p(y)$ 질의를 수행 가능하게 하는 방법을 소개한다. 이 방법은 A를 대각선(diagonal)들로 나누어, 각각에 대하여 기존의 1차원 *rank*와 *select* 함수에서 사용되었던 level directory 구조를 구축한다.

먼저 임의의 $p(i,j)$에 대한 $Diag_I(i,j)$, $Diag_{II}(i,j)$, $Diag_{III}(i,j)$, $Diag_{IV}(i,j)$를 각각 다음과 같이 정의한다. (그림 5)

- $Diag_I(i,j)$: $p(i,j)$를 포함하면서, 좌 상단에서 우 하단으로 이어지는 대각선 중 가장 긴 것
- $Diag_{II}(i,j)$: $p(i,j)$를 포함하면서, 우 상단에서 좌 하단으로 이어지는 대각선 중 가장 긴 것
- $Diag_{III}(i,j)$: $p(i,j)$를 포함하면서, 좌 하단에서 우 상단으로 이어지는 대각선 중 가장 긴 것
- $Diag_{IV}(i,j)$: $p(i,j)$를 포함하면서, 우 하단에서 좌 상단으로 이어지는 대각선 중 가장 긴 것

이제 자료구조 $DIAG_I(0..|Diag_I(i,j)|-1)$을 구축한다. 이 때, $DIAG_I[(i,j)]$에는 $Rect_I(i,j)$ 내에 포함된 1의 수를 기록한다. 따라서 $O(n)$개의 엔트리를 필요로 하고 하나의 엔트리가 $O(\log n^2)$ 비트를 사용하기 때문에, $DIAG_I((i,j))$ 하나가 사용하는 공간은 $O(n \log n)$ 비트이다. $DIAG_{II}$, $DIAG_{III}$, $DIAG_{IV}$도 $DIAG_I$과 동일한

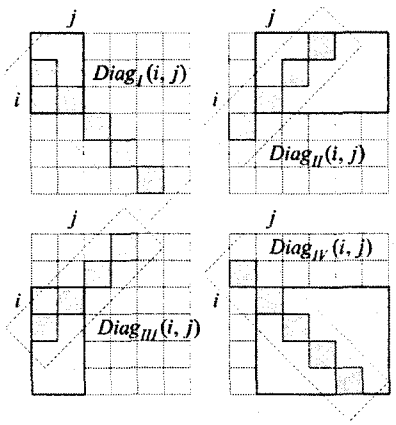

그림 5. $p(i,j)$에 대한 $Diag_I(i,j)$

방법으로 함께 구축한다. 또한, 중복되는 것을 제외하였을 때 A의 모든 $p(i,j)$에 대한 $Diag_I(i,j)$, $Diag_{II}(i,j)$, $Diag_{III}(i,j)$, $Diag_{IV}(i,j)$들은 각각 $2n-1$개씩 존재하므로, 모두 합쳐 $O(n)$개이다. 결국, A의 모든 $p(i,j)$에 대한 $DIAG_I$, $DIAG_{II}$, $DIAG_{III}$, $DIAG_{IV}$를 저장할 경우 $O(n^2 \log n)$ 비트를 필요로 하게 되며, 이는 우리가 3장에서 제안했던 자료구조들과 동일한 형태가 된다.

우리는 위에서 언급한 방법 대신, A의 모든 $p(i,j)$에 의해 정의되는 $2n-1$개의 $Diag_I(i,j)$들에 대하여 two-level directory 자료구조 $RANKDIR_I[0..2n-2]$를 구축한다. 임의의 $p(i,j)$의 $Diag_I(i,j)$ 하나에 대한 $RANKDIR_I[(i,j)]$는 다음과 같이 정의된다. (그림 6)

- 첫 번째 level directory : $Diag_I(i,j)$를 $\log^2 n$ 크기의 큰 블록(big block)들로 분할한다. 분할된 블록들 중 임의의 블록에 포함된 마지막 비트 위치를 $p(x,y)$라 할 때, 해당 블록에 대한 directory의 엔트리에는 $Rect_I(x,y)$ 내에 포함된 1의 수를 기록한다.

예를 들어 $1 \le i \le \lfloor |Diag_I(0,0)| / \log^2 n \rfloor$ 일 때, $Diag_I(0,0)$에 대한 첫 번째 level directory의 i번째 엔트리에는 $Rect_I(i\log^2 n, i\log^2 n)$ 내에 포함된 1의 수를 기록한다.

- 두 번째 level directory : $Diag_I(i,j)$를 $\log n$ 크기의 작은 블록(small block)들로 분할한다. 분할된 작은 블록들 중 임의의 블록에 포함된 마지막 비트를 $p(x,y)$, 해당 작은 블록이 포함되는 큰 블록의 직전 큰 블록에 포함된 마지막 비트를 $p(x',y')$라 할 때, 해당 작은 블록에 대한 directory의 엔트리에는 $Rect_I(x,y)$-$Rect_I(x',y')$의 영역 내에 포함된 1의 수를 기록한다.

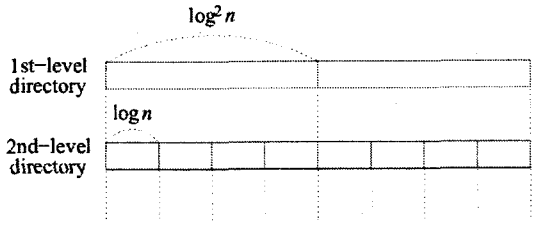

그림 6. $Diag_I(i,j)$에 대한 $RANKDIR_I[(i,j)]$

예를 들어 $1 \le i \le \lfloor |Diag_I(0,0)| / \log n \rfloor$ 이고 $i' = \lfloor i \log n / \log^2 n \rfloor$ 일 때, $Diag_I(0,0)$에 대한 두 번째 level directory의 i번째 엔트리에는 $Rect_I(i'\log^2 n + i\log n, i'\log^2 n + i\log n)$-$Rect_I(i'\log^2 n, i'\log^2 n)$의 영역 내에 포함된 1의 수를 기록한다.

$RANKDIR_I[(i,j)]$의 첫 번째 directory는 $O(\lfloor n/\log^2 n \rfloor)$개의 엔트리를 가지며, 하나의 엔트리는 $O(\log n^2)$ 비트를 필요로 한다. 따라서 첫 번째 directory를 저장하기 위해 필요한 공간은 $o(n)$ 비트이다. 하나의 $RANKDIR_I[(i,j)]$의 두 번째 directory는 $O(n/\log n)$개의 엔트리를 가지며, 하나의 엔트리는 $O(\log n + \log\log n)$ 비트를 필요로 한다. 따라서 두 번째 directory를 저장하기 위해 필요한 공간은 $O(n)$ 비트이다. 결국 $RANKDIR_I[(i,j)]$는 $O(n)$ 비트를 사용하며, A의 모든 $p(i,j)$에 대한 $RANKDIR_I[0..2n-2]$의 크기가 $O(n)$이므로, $RANKDIR_I[0..2n-2]$가 사용하는 공간은 $O(n^2)$ 비트이며, $O(n^2)$ 시간에 구축 가능하다. 또한, 자료구조 $RANKDIR_I[0..2n-2]$는 다음을 만족한다.

- $i \bmod (\log n) = 0$ 또는 $j \bmod (\log n) = 0$을 만족하는 임의의 $p(i,j)$에 대한 $Diag_I(i,j)$의 $RANKDIR_I[(i,j)]$는 첫 번째 directory의 $i/\log^2 n$번째 엔트리 값과 두 번째 directory의 $(i-((i/\log^2 n)\times\log^2 n))/\log n$번째 엔트리 값을 합하여 $Rect_I(i,j)$ 내의 1의 수를 $O(1)$ 시간에 반환한다. (3)

- $i \bmod (\log n) = 0$ 또는 $j \bmod (\log n) = 0$을 만족하고 $m < \log n$일 때, $Rect_I(i+m, j+m)$ 내에 포함된 1의 수는 $RANKDIR_I[(i,j)]$를 통해 $O(1)$ 시간에 얻은 $Rect_I(i,j)$ 내 1의 수와, $Rect_I(i+m, j+m)$- $Rect_I(i,j)$의 영역 내 1의 수의 합과 같다. (4)

이제 A의 n개의 row와 n개의 column들에 대해 1차원 $rank$ 자료구조 $rankdir_R[0..n-1]$과 $rankdir_C[0..n-1]$을 구축한다. 기존의 알고리즘들에 의해 $rankdir_R[i]$와

$rankdir_C[i]$는 $O(n)$ 비트에 구축될 수 있으므로, 이 자료구조가 사용하는 공간은 $O(n^2)$ 비트이며, 구축 시간은 $O(n^2)$이다. 또한, $rankdir_R[0..n-1]$와 $rankdir_C[0..n-1]$를 통하여 다음이 성립한다.

- 1차원 $rank$ 질의를 $2m(m < \log n)$번 수행하면 $Rect_I(i+m, j+m)-Rect_I(i, j)$ 내에 포함된 1의 수를 $O(\log n)$ 시간에 알 수 있다. (5)

우리는 (3), (4), (5)에 의해 $Rect_I(i+m, j+m)$ $(m < \log n)$ 내에 포함된 1의 수를 $O(\log n)$ 시간에 얻을 수 있다. 따라서, $RANKDIR_I[0..2n-2]$와 마찬가지로 $RANKDIR_{II}[0..2n-2]$, $RANKDIR_{III}[0..2n-2]$, $RANKDIR_{IV}[0..2n-2]$를 모두 구축하면, (2)에 의해 $O(\log n)$ 시간에 $Rank_p(x)$ 질의를 수행할 수 있다.

$Select_p(y)$ 질의의 경우 3장과 동일한 방법으로 이진 검색하여 $O(\log^2 n)$ 시간에 수행될 수 있다.

5. 결론

본 논문에서는 2차원 비트 스트링 내에서 동작하는 $Rank_p(x)$와 $Select_p(y)$ 함수를 정의하였다. 또한, $O(n^2 \log n)$ 비트의 자료구조를 $O(n^2)$ 시간에 구축하여 $Rank_p(x)$ 질의를 $O(1)$ 시간에, $Select_p(y)$ 질의를 $O(\log n)$ 시간에 수행하는 방법과, $O(n^2)$ 비트의 $RANKDIR_I[0..2n-2]$, $RANKDIR_{II}[0..2n-2]$, $RANKDIR_{III}[0..2n-2]$, $RANKDIR_{IV}[0..2n-2]$와 $rankdir_R[0..n-1]$과 $rankdir_C[0..n-1]$를 $O(n^2)$ 시간에 함께 구축하여 $Rank_p(x)$ 질의를 $O(\log n)$ 시간에, $Select_p(y)$ 질의를 $O(\log^2 n)$ 시간에 수행하는 방법을 제시하였다.

6. 향후 연구

$O(n^2)$ 비트의 2차원 비트 스트링 내에서 동작하는 $Rank_p(x)$와 $Select_p(y)$ 함수의 궁극적인 목표는 $o(n^2)$ 비트를 사용하여 $O(n^2)$ 시간에 구축된 자료구조를 기반으로 $O(1)$ 시간에 질의를 수행하는 것이 될 것이다. 효율적인 $Rank_p(x)$와 $Select_p(y)$ 함수는 향후 2차원 상에서의 압축된 인덱스 자료구조 개발이나 이미지 처리 분야 등에서 공간 효율을 위한 도구의 하나로 사용될 것이다.

참고문헌

[1] J. I. Munro and V. Raman, Succinct Representation of balanced parentheses and static trees, *SIAM J. Comput.*, 31(3):762-776, 2001

[2] G. Turan, Succinct representations of graphs, *Discrete Applied Math.*, 8:289-294, 1984

[3] G. Jacobson. Space-efficient static trees and graphs. *Proc. 30th IEEE Symp. Found. Computer Science*, 549-554, 1989

[4] G. Jacobson, *Succinct Static Data Structures*, PhD thesis, Carnegie Mellon University, Pittsburgh, 1988

[5] R. Raman and S. S. Rao, Succinct dynamic dictionaries and trees, *Proc. 30th Int. Colloq. Automata, Languages and Programming*, 357-368, 2003

[6] J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations, *Proc. 30th Int. Colloq. Automata, Languages and Programming*, 345-356, 2003

[7] J. I. Munro and S. S. Rao, Succinct representations of functions, *Proc. 31st Int. Colloq. Automata, Languages and Programming*, 1006-1015, 2004

[8] J. I. Munro, V. Raman, and S. S. Rao, Space efficient suffix trees. J. of Algorithms, 39(2):205-222, 2001

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

[10] K. Sadakane, Succinct representations of lcp information and improvements in the compressed suffix arrays, *Proc. 11th ACM-SIAM Symp. Discrete Algorithms*, 225-232, 2002

[11] 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*, 251-260, 2003

[12] R. Grossi, A. Gupta, and J. S. Vitter, High-order entropy-compressed text indexes, *Proc. 14th ACM-SIAM Symp. Discrete Algorithms*, 841-850, 2003

[13] J. C. Na, Linear-time construction of compressed su±x arrays using $O(n\log^{\epsilon} n)$-bit working space for large alphabets, *Proc. 16th Combinatorial Pattern Matching*, 57-67, 2005

[14] P. Ferragina and G. Manzini, Opportunistic data structures with applications, *Proc. 41st IEEE Symp. Found. Computer Science*, 390-398, 2000

[15] P. Ferragina and G. Manzini, An experimental study of an opportunistic index, *Proc. 12th ACM-SIAM Symp. Discrete Algorithms*, 269-278, 2001

[16] D. R. Clark, *Compact Pat Trees*, PhD thesis, University of Waterloo, Waterloo, 1996

[17] P. B. Miltersen, Lower bounds on the size of selection and rank indexes, *Proc. 16th ACM-SIAM Symp. Discrete Algorithms*, 11-12, 2005

[18] D. K. Kim, J. C. Na, J. E. Kim and K. Park, Fast Computation of Rank and Select Functions for Succinct Representation, *Algorithmica*, Submitted, 2006

[19] D. K. Kim and K. Park, Linear-Time Construction of Two-Dimensional Suffix Trees, *Proc. 26th Int. Colloq. Automata, Languages and Programming*, 463-472, 1999