블록정렬압축을 이용한 접미사배열의 효율적인 저장

(Efficient Storing of Suffix Arrays using Block-Sorting Compression)

이 건 호 † 박 근 수 ††

(Gunho Lee) (Kunsoo Park)

요 약 블록정렬압축은 빠른 속도로 동작하면서 높은 압축률을 나타내는 압축 방법이다. 또한 블록정렬방식으로 압축된 텍스트는 원래 텍스트를 복원하는 과정에서 접미사배열을 $O(n)$시간만에 구할 수 있다. 그러나 접미사배열을 이용하여 효율적인 검색을 수행하려면 lcp(longest common prefix)정보가 추가적으로 필요하다. 본 논문에서는 텍스트와 접미사배열이 주어졌을 때 lcp정보를 $O(n)$시간만에 구할 수 있는 알고리즘을 제시한다.

Abstract The block-sorting algorithm is a text compression method with good balance of compression ratio and speed. When a compressed text is decoded, the suffix array of the text is also obtained in $O(n)$ time. However, the longest common prefix (lcp) information is not obtained which is necessary for efficient searching with a suffix array. We present an algorithm to obtain the lcp information in $O(n)$ time from the original text and its suffix array.

1. 서 론

접미사배열(suffix array)[1]은 주어진 텍스트에서 각각의 접미사(suffix)들을 사전적 순서(lexicographical order)에 따라 정렬시킨 후 그 순서를 배열(array)형태로 가지고 있는 자료구조로, 텍스트에서 특정 패턴을 탐색하는 데에 효율적으로 사용될 수 있다. 텍스트의 길이가 n이고, 패턴의 길이가 m일 때, 접미사배열의 생성에는 $O(n\log n)$ 시간이 소요되고, 이를 이용한 검색에는 $O(m\log n)$시간이 소요된다. 그러나, longest common prefix(lcp)정보를 함께 가지고 있으면 $O(m+\log n)$시간에 검색을 수행할 수 있다. lcp정보는 일반적으로 접미사배열을 생성하면서 부가적으로 만들어진다. [1, 2]

블록정렬압축(Block Sorting Compression)[3, 4]은 압축률이 높고 실행 속도도 충분히 빠른 압축 방법으로, bzip2[5] 등의 압축 프로그램에 사용되고 있다. 특히 압축을 풀어 원문을 복원하는 데에는 선형시간만이 소요되며, 이 과정에서 원래 텍스트의 접미사배열 또한 부가적으로 생성할 수 있다[6].

블록정렬방식을 이용하여 텍스트를 압축하면 그 텍스트의 접미사배열도 함께 압축한 것과 같은 효과가 있기 때문에, 접미사배열의 저장이나 전송에 사용하기에 적합하다. 그러나, 접미사배열에서 효율적인 검색을 수행하기 위해 필요한 lcp정보는 별도로 구해야 한다. 블록정렬압축을 복원하면서 접미사배열을 얻어내는데 선형시간이 소요되기 때문에, 블록정렬압축을 이용하여 저장한 접미사배열을 효율적으로 이용하려면 lcp정보도 이에 비례한 시간 안에 구해낼 수 있어야 한다. 따라서 본 논문에서는 길이 n인 텍스트와 이 텍스트에 대한 접미사배열이 주어졌을 때 $O(n)$시간만에 필요한 lcp정보를 구할 수 있는 알고리즘을 제안하고자 한다.

논문의 구성은 다음과 같다. 2장에서는 접미사배열과 블록정렬압축, 그리고 블록정렬압축을 이용하여 접미사배열을 압축하는 방법에 대해 알아본다. 3장에서는 블록정렬압축을 이용하여 접미사배열을 압축하였을 때 해결해야 하는 문제에 대하여 살펴보고, 이를 해결하기 위한 알고리즘을 제시한다. 마지막으로 4장에서 결론을 맺는다.

· 이 논문은 2000년도 두뇌한국21사업에 의하여 지원되었음.

† 학생회원 : 서울대학교 컴퓨터공학부
ghlee@theory.snu.ac.kr
†† 종신회원 : 서울대학교 컴퓨터공학부 교수
kpark@theory.snu.ac.kr

논문접수 : 2000년 12월 19일
심사완료 : 2001년 6월 8일

2. 접미사배열과 블록정렬압축

2.1 접미사배열

주어진 긴 텍스트에서 패턴을 찾는 정보 검색에는 크게 두 가지가 있다. 첫번째는 패턴을 전처리하여 필요한 자료구조를 만든 후 텍스트를 보면서 검색을 수행하는 것으로, 문서 편집기에서 원하는 단어를 찾는 경우를 예로 들 수 있다. 두번째는 텍스트를 전처리하여 인덱스 자료구조를 만든 후 패턴을 보면서 검색을 수행하는 것으로, 책에서 색인을 보고 단어를 찾는 경우가 이에 해당한다. 두번째 유형에 사용되는 대표적인 자료구조로는 접미사나무와 접미사배열이 있다[2].

Manber와 Myers에 의해 제안된 접미사배열[1]은 각각의 접미사들을 사전적 순서에 따라 정렬시킨 후 그 순서를 배열형태로 가지고 있는 자료구조이다. 접미사배열은 접미사나무에 비해 생성시간이 오래 걸리는 대신 저장공간을 적게 사용하고 구조가 간단한 실용적인 모델이다. $A=a_1a_2..a_n$을 길이 n의 텍스트라고 할 때, $A_i=a_ia_{i+1}...a_n$을 i=1, ... , n 위치에서 시작하는 A의 접미사라라고 하자. 예를 들어 A=*abaababaabaab*라 할 때, A의 접미사들은 다음과 같다.

A_1=*abaababaabaab*, A_2=*baababaabaab*, ... , A_{13}=*b*.

접미사배열의 기본적인 자료구조는 A의 모든 접미사들을 사전적 순서로 정렬했을 때의 순서를 저장한 배열 Pos이다. 즉, 집합 $\{A_1, A_2, ..., A_n\}$ 중에서 k번째로 작은 접미사가 A_i이면 $Pos[k]=i$ 이다.

Pos배열이 있으면, 이진 탐색 방법을 사용하여 텍스트 A에서 길이 m인 패턴 P를 검색하는데 $O(m\log n)$시간이 소요된다. Pos배열과 함께 특정한 접미사들간의 lcp정보를 부가적으로 이용하면 이진 탐색 방법을 약간 변형하여 길이가 m인 패턴을 $O(m+\log n)$시간에 검색할 수 있다.

Pos배열은 사전적 순서로 정렬되어 있기 때문에, 공통된 접두사(prefix)를 가지는 접미사들은 Pos배열상에서 연속된 위치에 나타나게 된다. 즉 A_i와 A_j 가 둘 다 접두사 B를 갖는다면, Pos배열 상에서 A_i와 A_j사이에 있는 접미사들은 모두 B를 접두사로 가지게 된다. 이를 이용하면 이진탐색시 비교하는 문자의 수를 줄일 수 있게 된다.

이진탐색시 한 검색 구간에서의 왼쪽 끝점을 L, 오른쪽 끝점을 R, 그리고 L과 R의 중앙점을 M이라고 하자. 또한 두 문자열 X와 Y에서 가장 긴 공통 접두사의 길이를 $lcp(X, Y)$라 표현한다. 구간 (L, R)에서는 A_M과 P를 비교하게 되고, 이것이 실패하면 다음 검색 구간으로 넘어가게 된다. P가 A_M보다 사전적으로 앞에 있다면 다음 검색구간은 (L, M)이 되고, L과 M의 중앙점을 M'이라 할 때 P를 $A_{M'}$과 비교하게 된다. 이 때, $lcp(A_L, A_M)$을 알고 있다면 A_M과 $A_{M'}$이 적어도 $lcp(A_L, A_M)$만큼의 공통된 접두사를 가지고 있음을 알 수 있으므로, 그 이후의 문자부터 비교해나가는 것으로 충분하다. 다음 검색구간이 (M, R)이 될 때에도 $lcp(A_M, A_R)$를 알고 있다면 같은 방법을 사용할 수 있다.

길이 n의 텍스트 전체에 대해서 이진탐색을 수행할 때, 가능한 구간과 중간점의 triple(L, M, R)들의 집합을 고려해보자. 여기서 $1\le L<M<R\le n$ 이고, $M\in[2, n-1]$이다. 이때, $n-2$개의 중앙점 M 각각에 대하여 (L, R)은 유일하게 결정되므로, 한 중앙점 M에 대한 triple (L, M, R)을 (L_M, M, R_M)이라고 표시할 수 있다. n=7일 때 이진 탐색의 경로와 가능한 (L_M, M, R_M)들의 집합을 트리 모양으로 도시하면 그림 1과 같다. 그림 1에서 내부 노드로 표시되는 것이 가능한 (L_M, M, R_M)들이다. 단말노드로 표시되는 것은 이진탐색의 마지막 단계로, 인접한 문자열을 비교하는 경우이다.

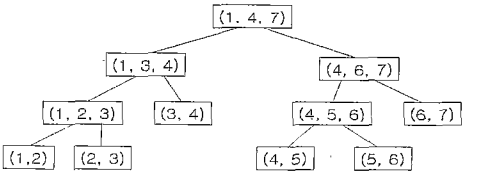

그림 1 접미사배열에서의 이진탐색트리

효율적인 이진탐색을 위해 필요한 lcp정보들은 다음과 같이 $Llcp$, $Rlcp$ 두 개의 배열로 표현할 수 있다.

$$Llcp[M] = lcp(A_{Pos[L_M]}, A_{Pos[M]})$$
$$Rlcp[M] = lcp(A_{Pos[M]}, A_{Pos[R_M]})$$

즉, 필요한 lcp정보는 $O(n)$개이다.

표 1은 A=*abaababaabaab*일 때 Pos배열과 $Llcp$, $Rlcp$ 배열의 예이다.

표 1 A=*abaababaabaab*일 때 Pos, $Llcp$, $Rlcp$ 배열

k	$Pos[k]$	$A_{Pos[k]}$	L_k	$Llcp[k]$	R_k	$Rlcp[k]$
1	11	*aab*		-		-
2	8	*aabaab*	1	3	3	4
3	3	*aababaabaab*	1	3	4	1
4	12	*ab*	1	1	7	2
5	9	*abaab*	4	2	6	5
6	6	*abaabaab*	4	2	7	6
7	1	*abaababaabaab*	1	1	13	0
8	4	*ababaabaab*	7	3	9	0
9	13	*b*	7	0	10	1
10	10	*baab*	7	0	13	2
11	7	*baabaab*	10	4	12	5
12	2	*baababaabaab*	10	4	13	2
13	5	*babaabaab*		-		-

2.2 블록정렬압축

비손실압축에는 크게 두 가지 접근방식이 있다. 하나는 통계적 방식이고, 또 하나는 사전적 방식이다. 일반적으로 통계적 방식은 뛰어난 압축률을 보이지만 수행속도가 상당히 느린 반면, 사전적 방식은 압축률은 약간 떨어지는 대신 수행속도가 월등히 빠른 특징을 가지고 있다[3].

블록정렬압축은 통계적 방식에 버금가는 압축률을 보이면서 사전적 방식에 비견될만한 빠른 속도로 동작하는 압축 방법이다. 특히 원문을 복원하는 데는 선형시간이 소요되며, 실제 구현했을 때의 수행 속도도 충분히 빠르다[3, 4]. 블록정렬압축방식은 bzip2[5] 등의 압축 프로그램에 사용되고 있다. 블록정렬압축은 다음과 같은 순서로 진행된다.

① Burrows-Wheeler Transform
② Move-To-Front coding
③ Entropy coding(Huffman, run-length encoding, etc...)

첫 번째 단계인 Burrows-Wheeler Transform(BWT)은 입력된 텍스트 A를 회전시켜서 만들 수 있는 모든 텍스트를 정렬하였을 때, 원래 텍스트의 위치 I와 맨 마지막 문자들로 이루어진 새로운 텍스트 L을 출력한다. 예를 들어 A='*abraca*'일 때, BWT의 수행 결과는 그림 2와 같이 I=2, L='*caraab*'이다.

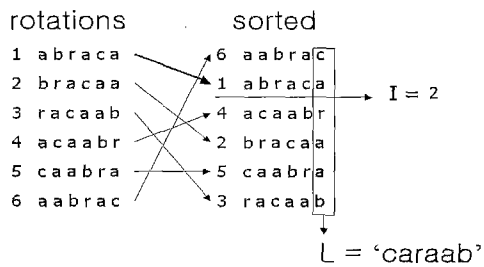

그림 2 Burrows-Wheeler Transform (BWT)

BWT는 단지 문자들의 순서를 바꿀 뿐 압축을 하지는 않는다. 그러나, BWT의 결과로 나온 텍스트 L은 지역성(locality)이 높아지게 된다. 즉, 원래 텍스트 A에 비해 같은 문자들이 비슷한 곳에 집중적으로 나타나게 된다.

지역성이 높아지는 이유를 직관적으로 살펴보기 위해 'the'라는 단어를 많이 포함하고 있는 영어로 된 텍스트를 예로 들어보자. 회전시킨 텍스트들을 정렬했을 때 'he '로 시작하는 텍스트들을 살펴보면, 마지막 문자는 't'가 대부분일 것이다. 영어 문장의 특성상 't'외에 나올만한 문자는 'T', 's', 'S', ' ' 정도이다. 이와 같이 BWT의 결과로 출력된 텍스트 L에서는 적은 종류의 문자들이 비슷한 위치에 집중적으로 나타나게 된다.

두 번째 단계인 Move-To-Front coding은 텍스트 L을 입력으로 받아, 특정한 위치에 있는 문자 *ch*를, 그 위치의 *ch*와 그 직전에 나타난 *ch* 사이에 존재하는 서로 다른 문자의 개수로 부호화(encoding)한 벡터 R을 출력한다. 문자의 개수를 세기 위해서 리스트 Y를 유지하게 되는데, 초기상태에서 Y는 L에 사용된 문자들이 한번씩만 나타나도록 초기화되며, 인덱스는 0부터 시작한다. L의 처음부터 한 문자씩 보면서 현재 Y에서의 그 문자의 위치를 R로 출력하고, Y에서는 그 문자를 맨 앞으로 이동시킨다. 그림 3은 Y=[*a*, *b*, *c*, *r*]로 초기화되고, L='*caraab*'가 입력으로 들어왔을 때 Move-To-Front coding의 수행 과정을 나타낸다.

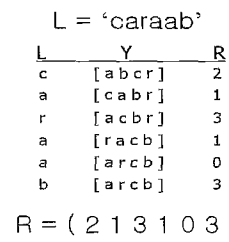

그림 3 Move-To-Front coding

첫 번째 단계에서 BWT의 결과로 출력된 텍스트 L은 문자의 지역성이 높기 때문에 Move-To-Front coding을 거친 벡터 R에는 작은 수의 빈도가 높게 나타난다. 특히 0의 경우 빈도가 압도적으로 높고, 연속되서 나타나는 경우가 많아진다. 따라서 세 번째 단계에서 벡터 R에 Huffman encoding이나 run-length encoding을 적용하면 높은 압축률을 얻을 수 있게 된다.

압축을 풀기 위해서는 위의 과정을 다음과 같이 거꾸로 수행하면 된다.

① Entropy decoding
② Move-To-Front decoding
③ Reverse-BWT

압축할 때 Entropy coding에 사용한 방법에 따라 달라질 수 있지만, 일반적으로 사용하는 Huffman encoding이나 run-length encoding의 경우에는 $O(n)$시간에 역부호화(decoding)가 가능하다. Move-To-Front deco-

ding이나 Reverse-BWT도 모두 $O(n)$시간에 역으로 변환하는 것이 가능하다. 따라서 블록정렬방식으로 압축된 텍스트에서 원래 텍스트를 복원하는 데는 일반적으로 선형시간만이 소요된다.

2.3 블록정렬방법을 이용한 접미사배열의 압축

블록정렬압축의 첫번째 단계인 BWT를 수행하기 위해서는 회전시켜서 만들 수 있는 텍스트들을 정렬해야 하는데, 이는 접미사배열의 생성과정과 유사하다. 입력된 텍스트 A의 마지막에 사전적 순서가 가장 빠른 단말기호가 붙어있고, A에 대한 접미사배열 Pos를 알고 있다면, 이를 이용하여 L과 I를 쉽게 구할 수 있다. 즉, $L[k]$는 A를 회전시켜서 만든 텍스트들 중 k번째 텍스트의 마지막 문자인데, 이는 k번째로 작은 접미사의 바로 앞에 위치하는 문자와 같다. 즉, $L[k]$는 $a_{Pos[k]-1}$이다. 단, $Pos[k]$가 1이라면 회전시키지 않은 텍스트가 k번째로 정렬된다는 뜻이므로, 이때 $L[k]$는 A의 마지막 문자이고, I는 k가 된다. 그림 4는 위와 같은 방법으로 BWT를 수행하는 알고리즘이다.

```
Algorithm BWT
1   for k:= 1 to n do
2           j:=Pos[k]-1
3           if j=0 then j:=n, I:=k  fi
4           L[k]:=A[j]
5   od
```

그림 4 Algorithm BWT

```
Algorithm Reverse-BWT
1   C[for all characters]:=0
2   for k:=1 to n do
3           C[L[k]]:=C[L[k]]+1
4           P[k]:=C[L[k]]
5   od
6   sum:=0
7   for ch:=each character do
8           sum:=sum+C[ch]
9           C[ch]:=sum-C[ch]
10  od
11  k:=I
12  for j:=n to 1
13          Pos[k]:=j+1
14          if Pos[k]=n+1 then Pos[k]:=1 fi
15          A[j]:=L[k]
16          i:=P[k]+C[L[k]]
17  od
```

그림 5 Algorithm Reverse-BWT

마찬가지로 BWT의 출력 결과인 I와 L을 알고 있다면, 그림 5의 알고리즘으로 원래 텍스트 A와 이에 대한 접미사배열 Pos를 구할 수 있다. 그림 5의 알고리즘에서 접미사배열 Pos를 구하기 위해 추가한 부분은 13번과 14번 뿐이다. 즉, 블록정렬방식으로 압축된 텍스트에서 원래 텍스트를 복원해내는 과정에서 시간복잡도의 변화 없이 접미사배열 Pos도 부가적으로 구해낼 수 있다[6].

따라서 블록정렬방식을 이용하여 텍스트를 압축하면 그 텍스트의 접미사배열도 함께 압축한 것과 같은 효과가 있기 때문에, 접미사배열의 저장이나 전송에 사용하기에 적합하다. K.Sadakane와 H.Imai는 [6]에서 분산 텍스트 데이타베이스 관리에 이 방법을 도입하여, 수집된 데이타를 보내는데 필요한 통신량을 줄임과 동시에 인덱싱을 위한 부담을 분산시키는 방법을 제안하였다. 또한 K.Sadakane는 [7]에서 BWT를 변형하여 Case-insensitive 검색에 활용할 수 있는 접미사배열을 생성하는 방법을 제시하였다.

3. lcp 정보를 구하는 알고리즘

접미사배열을 이용하여 효율적인 검색을 수행하려면 lcp 정보가 필요하다. 텍스트에서 직접 접미사배열을 생성하는 경우에는 $O(n\log n)$시간을 사용하여 접미사배열과 lcp정보를 함께 구해낼 수 있지만[1, 2], 블록정렬방식으로 압축된 텍스트에서 접미사배열을 생성하는 경우에는 $O(n)$시간에 원래 텍스트와 접미사배열만을 구해낼 수 있다. 따라서, 본 논문에서는 텍스트와 접미사배열이 주어졌을 때 $O(n)$시간만에 lcp정보를 구할 수 있는 알고리즘을 제안하고자 한다.

3.1 문제 정의

주어진 텍스트 $A = a_1a_2..a_n$ 의 접미사들의 집합 $\{A_1, A_2,..., A_n\}$을 사전적 순서로 정렬했을 때, k번째로 작은 접미사가 A_i이면 $Pos[k]=i$이고 $Rank[i]=k$라고 정의한다. 즉, $i = Pos[Rank[i]]$이다. 또한, Pos배열 상에서 인접한 접미사들 간의 lcp값을 다음과 같이 정의한다 : $2\le k\le n$에 대하여 $height[k] = lcp(A_{Pos[k-1]}, A_{Pos[k]})$. 압축된 텍스트로부터 원래 텍스트를 복원하는 과정에서 접미사배열도 함께 생성할 수 있으므로, Pos와 $Rank$, A는 이미 알려져 있다고 가정할 수 있다. 또한 접미사배열을 이용하여 $O(m+\log n)$시간만에 검색을 수행하는데 필요한 lcp정보($Llcp$, $Rlcp$)는 $height[2..n]$로부터 $O(n)$ 시간에 구할 수 있다[1, 2]. 따라서 아래에서는 Pos, $Rank$, A가 주어졌을 때 $height[2..n]$을 $O(n)$시간

에 구하는 방법에 대해서 생각해보도록 한다.

3.2 lcp의 성질

*Pos*배열은 사전적 순서로 정렬되어 있다. 따라서 *Pos*배열상에서 떨어져 있는 두 접미사간의 lcp는 그들 사이에 있는 인접한 접미사들간의 lcp값들 중 최소값을 갖게 된다[1]. 즉,

$$lcp(A_{Pos[w]}, A_{Pos[y]}) = min(lcp(A_{Pos[x]}, A_{Pos[x+1]})),\ w \le x < y. \quad (1)$$

이다. (1)에 따르면 *Pos*배열상에서 인접한 접미사간의 lcp는, 그들을 사이에 두고 떨어져 있는 두 접미사간의 lcp보다 크거나 같아야 하므로 다음 성질도 마찬가지로 만족한다.

$$lcp(A_{Pos[x]}, A_{Pos[x+1]}) \ge lcp(A_{Pos[w]}, A_{Pos[y]}),\ w \le x < y \quad (2)$$

(2)에 의해 $w=x$일 때 다음 참고 1이 성립한다.

참고 1

$x < y$에 대하여, $lcp(A_{Pos[x]}, A_{Pos[x+1]}) \ge lcp(A_{Pos[x]}, A_{Pos[y]})$ 이다.

또한 사전적으로 인접한 두 접미사간의 lcp가 1보다 클 경우, 각각의 접미사에서 맨 첫 번째 문자를 삭제한 접미사들 사이에서도 사전적 순서는 여전히 유지된다. 즉 $lcp(A_{Pos[x]}, A_{Pos[x+1]}) > 1$일 경우, $A_{Pos[x]+1}$이 $A_{Pos[x+1]+1}$ 보다 사전적으로 앞서있으므로 다음 참고 2가 성립한다.

참고 2

$lcp(A_{Pos[x]}, A_{Pos[x+1]}) > 1$일 때, $Rank[Pos[x]+1] < Rank[Pos[x+1]+1]$ 이다.

이때 $A_{Pos[x]+1}$과 $A_{Pos[x+1]+1}$사이의 lcp는 원래 접미사간의 lcp에서 1을 뺀 값이 되므로 참고 3도 성립한다.

참고 3

$lcp(A_{Pos[x]}, A_{Pos[x+1]}) > 1$ 일 때, $lcp(A_{Pos[x]+1}, A_{Pos[x+1]+1}) = lcp(A_{Pos[x]}, A_{Pos[x+1]})-1$ 이다.

순서	위치
...	
p a b c b d a b e	*i* = *Pos*[*p*]
p+1 a b e	*j* = *Pos*[*p*+1]
...	
q b c b d a b e	*i*+1 = *Pos*[*q*] = *Pos*[*p*]+1
q+1 b d a b e	
...	
... b e	*j*+1 = *Pos*[*p*+1]+1
...	

그림 6 정렬된 접미사와 height의 예

이제 *Pos*배열상에서 인접한 임의의 접미사 A_i와 A_j의 lcp를 알고 있을 때, A_{i+1}과 A_{i+1}에 인접한 접미사간의 lcp를 구하는 방법에 대해서 생각해보도록 한다. 그림 6에서와 같이 $p = Rank[i]$, $q = Rank[i+1]$이라고 하자. 이때, 다음 보조정리 1이 성립한다.

보조정리 1.

$lcp(A_{Pos[p]}, A_{Pos[p+1]}) > 1$일 때, $lcp(A_{Pos[q]}, A_{Pos[q+1]}) \ge lcp(A_{Pos[q]}, A_{Pos[p+1]+1})$ 이다.

증명.

참고 2에 의해

$$q = Rank[i+1] = Rank[Pos[p]+1] < Rank[Pos[p+1]+1] \quad (3)$$

(3)과 참고 1에 의해 $lcp(A_{Pos[q]}, A_{Pos[q+1]}) \ge lcp(A_{Pos[q]}, A_{Pos[Rank[Pos[p+1]+1]]})$. $Pos[Rank[Pos[p+1]+1]] = Pos[p+1]+1$이므로 $lcp(A_{Pos[q]}, A_{Pos[q+1]}) \ge lcp(A_{Pos[q]}, A_{Pos[p+1]+1})$ 이다. □

즉, A_{i+1}과 A_{i+1}에 인접한 접미사간의 lcp는 A_{i+1}과 A_{j+1}간의 lcp보다 크거나 같다. 따라서 다음과 같이 정리 1이 성립한다.

정리 1.

$height[Rank[i]+1] > 1$일 때, $height[Rank[i+1]+1] \ge height[Rank[i]+1] - 1$이다.

증명.

$$\begin{aligned} & height[Rank[i+1]+1] \\ &= height[q+1] \\ &= lcp(A_{Pos[q]}, A_{Pos[q+1]}) \\ &\ge lcp(A_{Pos[q]}, A_{Pos[p+1]+1}) && (\because \text{보조정리 1}) \\ &= lcp(A_{Pos[p]+1}, A_{Pos[p+1]+1}) \\ &= lcp(A_{Pos[p]}, A_{Pos[p+1]}) - 1 && (\because \text{참고 3}) \\ &= height[p+1] - 1 \\ &= height[Rank[i]+1] - 1 \end{aligned}$$ □

정리 1에 의하면 A_i와 A_i에 인접한 접미사 사이의 lcp가 h일 때, A_{i+1}와 A_{i+1}에 인접한 접미사는 첫번째 문자부터 h-1번째 문자까지가 같다는 것이 보장된다. 따라서 A_{i+1}와 A_{i+1}에 인접한 접미사 사이의 lcp를 구할 때는 h번째 문자부터 비교하는 것으로 충분하다. h가 1 이하라면 첫번째 문자부터 비교해나가면 된다.

3.3 알고리즘 및 분석

2절에서 살펴본 바와 같이 임의의 접미사 A_i와 A_i에 인접한 접미사 사이의 lcp를 알고 있다면, A_{i+1}과 A_{i+1}에 인접한 접미사 사이의 lcp를 구할 때는 모든 문자를 비교해보지 않아도 된다. 따라서 접미사 A_1부터 A_n까지 차례로 *Pos*배열상에서 인접한 접미사들과의 lcp를 구해나가면 모든 인접한 접미사들간의 lcp를 효율적으로 구

할 수 있다. 그림 7은 이러한 방법으로 *height*[2..*n*]를 구하는 알고리즘이다.

```
Algorithm GetHeight
 1   cp:=0
 2   for i:=1 to n do
 3       if Rank[i]+1<=n then
 4           j:=Pos[Rank[i]+1]
 5           while ( (i+cp<=n ) and (j+cp<=n)
             and (a_{i+cp}=a_{j+cp}) ) do
 6               cp:=cp+1
 7           od
 8           height[Rank[i]+1] = cp
 9           if (cp>0) then cp:=cp-1  fi
 10      fi
 11  od
```

그림 7 Algorithm GetHeight

정리 2.

Algorithm GetHeight는 height배열을 $O(n)$시간에 계산한다.

증명.

정리 1과 위의 논의에 의해 Algorithm GetHeight가 height배열을 올바로 계산하고 있음을 알 수 있다. 이 알고리즘의 실행 시간은 루프의 가장 안쪽에 있는 6번 문장의 실행 횟수에 비례한다. 6번문장에서는 cp의 값이 하나씩 증가하는데, 이는 5번 문장의 조건에 의해 $cp < n$일 때만 실행된다. 2-11번 사이의 루프는 n번 반복되므로, 9번 문장에서 cp의 값은 n보다 많이 감소할 수 없고, cp의 초기치는 0이므로, cp는 $2n$ 이상 증가할 수 없다. 따라서 6번문장은 $2n$번 이상 수행될 수 없다. 그러므로 Algorithm GetHeight의 시간복잡도는 $O(n)$이다. □

2절에서 언급한 바와 같이 인접한 모든 접미사들간의 lcp, 즉 *height*[2..*n*]를 알고 있으면 접미사배열을 이용한 검색에 필요한 모든 lcp 정보를 $O(n)$시간에 구할 수 있다. 따라서 Algorithm GetHeight를 사용하면 접미사배열과 텍스트가 주어졌을 때, 접미사배열을 이용한 효율적인 검색에 필요한 lcp정보를 $O(n)$시간에 구할 수 있다.

4. 결 론

본 논문에서는 접미사배열과 텍스트가 주어졌을 때 $O(n)$시간만에 접미사배열을 이용한 효율적인 검색에 필요한 lcp정보를 구하는 알고리즘을 제시하였다. 블록정렬압축을 사용하면 접미사배열도 함께 압축한 것과 같은 효과를 얻을 수 있고, 본 논문에서 제시한 알고리즘을 사용하면 검색에 필요한 lcp정보도 효율적으로 구해낼 수 있다. 따라서 블록정렬압축을 사용하여 접미사배열을 효율적으로 저장하고 사용하는 것이 가능하다.

참 고 문 헌

[1] U. Manber and G. Myers, "Suffix arrays : a new method for on-line string searches," *SIAM Journal on Computing*, Vol.22, No.5, pp.935-948, 1993.

[2] 이시은, 박근수, "Suffix Array를 구축하는 새로운 알고리즘", 정보과학회논문지(A), 제24권, 제7호, pp.697-704, 1997.

[3] M. Burrows and D. J. Wheeler, "A Block-sorting Lossless Data Compression Algorithm," *Digital Systems Research Center Research Report* 124, 1994.

[4] P. Fenwick, "Block Sorting Text Compression," *Australian Computer Science Communications*, Vol.18, No.1, pp.193-202, 1996.

[5] J. Seward, http://sources.redhat.com/bzip2

[6] K. Sadakane and H. Imai, "A Cooperative Distributed Text Database Management Method Unifying Search and Compression Based on the Burrows-Wheeler Transformation," *Proc. of International Workshop on New Database Technologies for Collaborative Work Support and Spatio-Temporal Data Management* (*NewDB'98*), pp.434-445, 1998.

[7] K. Sadakane, "A Modified Burrows-Wheeler Transformation for Case-insensitive Search with Application to Suffix Array Compression," *Proc. of Data Compression Conference* (*DCC'99*), p.548, 1999.

이 건 호

1999년 서울대학교 컴퓨터공학과 학사. 2001년 서울대학교 컴퓨터공학부 석사. 관심분야는 Design and analysis of Algorithms

박 근 수

1983년 서울대학교 컴퓨터공학과 학사. 1985년 서울대학교 컴퓨터공학과 석사. 1991년 미국 Columbia University 전산학 박사. 1991년 ~ 1993년 영국 University of London, King's College 조교수. 1993년 ~ 현재 서울대학교 컴퓨터공학부 부교수. 관심분야는 컴퓨터이론, 암호학, 병렬계산