블록정렬압축을 이용한 접미사배열의 효율적인 저장

이건호[U] 박근수
서울대학교 컴퓨터공학부
{ghlee, kpark}@theory.snu.ac.kr

Efficient storing of suffix arrays using block-sorting compression

Gunho Lee[U] Kunsoo Park
School of Computer Science and Engineering, Seoul National University

요 약

블록정렬압축은 빠른 속도로 동작하면서 높은 압축률을 나타내는 압축 방법이다. 또한 블록 정렬방식으로 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.

1. 서론

본 논문에서는 텍스트와 접미사배열이 주어졌을 때 $O(n)$시간만에 접미사배열을 이용해서 효율적인 검색을 수행하는데 필요한 lcp 정보를 구할 수 있는 알고리즘을 제안한다. 이 알고리즘은 블록정렬압축을 사용하여 압축된 텍스트에서 접미사행렬을 생성하여 효과적인 검색을 수행하는데 사용될 수 있다.

2장에서는 접미사배열과 블록정렬압축에 대해 살펴보고, 3장에서는 알고리즘을 제시하며 4장에서 결론을 맺는다.

2. 접미사배열과 블록정렬압축

2.1 접미사배열

주어진 긴 텍스트에서 패턴을 찾는 정보 검색에는 크게 두가지가 있다. 첫번째는 패턴을 전처리하여 필요한 자료구조를 만든 후 텍스트를 보면서 검색을 수행하는 것이고, 두번째는 텍스트를 전처리하여 인덱스 자료구조를 만든 후 패턴을 보면서 검색을 수행하는 것이다. 두번째 유형에 해당하는 대표적인 자료구조로는 접미사나무(Suffix tree)와 접미사배열(Suffix array)이 있다[6].

접미사배열[1]은 각각의 접미사들을 사전적 순서(lexicographical order)에 따라 정렬시킨 후 그 순서를 배열형태로 가지고 있는 자료구조이다. 접미사배열은 접미사나무에 비해 생성시간이 오래 걸리는 대신 저장공간을 적게 사용하고 구조가 간단한 실용적인 모델이다.

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, lcp (longest common prefix)정보를 함께 가지고 있으면 $O(m+\log n)$ 시간에 검색을 수행할 수 있다. lcp정보는 일반적으로 접미사배열을 생성하면서 부가적으로 만들어진다.

2.2 블록정렬압축

비손실압축에는 크게 두가지 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 복원하는데는 선형시간이 소요되며, 실제 구현했을

때의 수행 속도도 충분히 빠르다[3, 5]. 블록정렬압축방식은 bzip2[7] 등의 압축프로그램에 사용되고 있다.

입력 텍스트가 A라고 할 때, 블록정렬압축은 대략적으로 다음과 같이 진행된다.

↓ A (input text)
· Burrows-Wheeler Transform (BWT)
↓ (L, I)
· Encoding
→ (OUT, I)

이중 첫번째 단계인 BWT는 입력된 텍스트 A를 회전시켜서 만들 수 있는 모든 텍스트를 정렬하였을 때 원래 텍스트의 위치 I와 맨 마지막 문자들로 이루어진 텍스트 L을 추출하는 과정이다. 예를 들어 A='abrac$'일 때 BWT의 수행 결과는 그림 1과 같다.

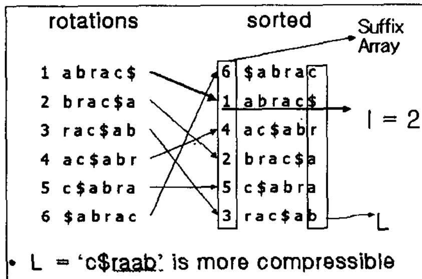

그림 1. Burrows-Wheeler Transform

2.3 블록정렬방법을 이용한 접미사배열의 압축

블록정렬압축의 첫번째 단계인 BWT를 수행하기 위해서는 압축할 텍스트의 접미사들, 혹은 회전시켜서 만들 수 있는 텍스트들을 정렬해야 하는데, 이는 접미사배열의 생성과정과 유사하다. 이를 이용하면, 압축된 텍스트에서 원래 텍스트를 복원하는 과정에서 접미사배열도 역시 $O(n)$시간만에 구할 수 있다.

따라서 블록정렬방식을 이용하여 텍스트를 압축하면 그 텍스트의 접미사배열도 함께 압축한 것과 같은 효과가 있기 때문에, 접미사배열의 저장이나 전송에 사용하기에 적합하다[2, 4].

2.4 lcp정보의 계산

접미사배열을 이용하여 효율적인 검색을 수행하려면 lcp 정보가 필요하다. 기존의 방법은 $O(n\log n)$시간을 사용하여 접미사배열을 생성하는 과정에서 부가적으로 lcp 정보를 구해낸다[1, 6].

그러나, 블록정렬방식으로 압축된 텍스트에서는 $O(n)$시간만에 원래 텍스트와 접미사배열을 구해낼 수 있으므로, 접미사배열을 생성하면서 부가적으로 lcp정보를 얻는 기존의 방법을 사용하는 것은 비효율적이다.

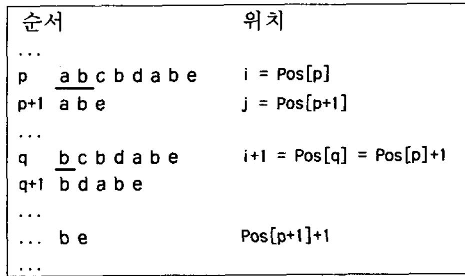

그림 2. 정렬된 접미사와 height의 예

따라서 본 논문에서는 텍스트와 접미사배열이 주어졌을 때 $O(n)$시간만에 lcp정보를 구할 수 있는 알고리즘을 제안한다.

3. 알고리즘

3.1 정의

문자를 구성하는 알파벳이 Σ이고, n개의 문자로 이루어진 텍스트 $A = a_1a_2...a_n$ ($a_i \in \Sigma$, $1 \le i \le n$)이 있을 때, i번째 접미사를 $A_i = a_ia_{i+1}...a_n$라 표시한다. 접미사들의 집합 $\{A_1, A_2, ..., A_n\}$을 사전적 순서로 정렬했을 때, k번째 접미사의 시작위치를 Pos[k], A_k의 순위를 Rank[k]라 한다. 즉, k = Pos[Rank[k]]이다. 또한, 두 텍스트 T_1, T_2에 대하여 lcp(T_1, T_2)는 T_1과 T_2의 공통된 접두사의 길이를 나타내고, height[k] = lcp($A_{Pos[k-1]}$, $A_{Pos[k]}$) 라 정의한다.

3.2 lcp의 관찰

압축된 텍스트에서 원래 텍스트를 복원하는 과정에서 텍스트와 접미사배열이 생성되므로 Pos와 Rank, A는 이미 알려져 있다.

접미사배열을 이용하여 $O(m+\log n)$시간만에 검색을 수행하는데 필요한 lcp정보는 height[2..n]로부터 $O(n)$ 시간에 구할 수 있으므로[1, 6], 아래에서는 height[2..n]을 $O(n)$시간에 구하기 위한 방법에 대해서 생각해보도록 한다.

[1]에 의하면,

$$\text{lcp}(A_{Pos[x]}, A_{Pos[y]}) = \min(\text{lcp}(A_{Pos[z]}, A_{Pos[z+1]})), \; x \le z \le y-1$$

이므로,

$$\text{lcp}(A_{Pos[x]}, A_{Pos[x+1]}) \ge \text{lcp}(A_{Pos[x]}, A_{Pos[y]}), \; x < y \quad \cdots ①$$

이다.

또한 사전적으로 인접한 두 접미사간의 lcp 가 1보다 클 경우, 각각의 접미사에서 맨 첫 번째 문자를 삭제한 접미사들 사이에서도 사전적 순서는 여전히 유지되며, 이들간의 lcp는 원래 접미

사간의 lcp에서 1을 뺀 값이 된다. 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

$$Rank[Pos[x]+1] < Rank[Pos[x+1]+1] \quad \cdots ②$$

이며,

$$lcp(A_{Pos[x]+1}, A_{Pos[x+1]+1}) = lcp(A_{Pos[x]}, A_{Pos[x+1]})-1 \quad \cdots ③$$

도 성립한다.

이후로 편의상 p = Rank[i], q = Rank[i+1] 이라고 하자. height[p+1] > 1 일 경우 ②에 의해

$$Rank[Pos[p]+1] < Rank[Pos[p+1]+1]$$
$$\leftrightarrow Rank[i+1] < Rank[Pos[p+1]+1]$$
$$\leftrightarrow q < Rank[Pos[p+1]+1]$$

이므로, ①에 의해

$$lcp(A_{Pos[q]}, A_{Pos[q+1]})$$
$$\geq lcp(A_{Pos[q]}, A_{Pos[Rank[Pos[p+1]+1]]})$$
$$= lcp(A_{Pos[q]}, A_{Pos[p+1]+1}) \quad \cdots ④$$

이 성립한다.

위와 같은 관찰에 의해 다음 정리가 성립함을 알 수 있다.

정리 1.
height[Rank[i]+1] > 1 일 때,
height[Rank[i+1]+1]
≥ height[Rank[i]+1] - 1

증명.

$$height[Rank[i+1]+1] = height[q+1]$$
$$= lcp(A_{Pos[q]}, A_{Pos[q+1]})$$
$$\geq lcp(A_{Pos[q]}, A_{Pos[p+1]+1}) \quad (\because ④)$$
$$= lcp(A_{Pos[p]+1}, A_{Pos[p+1]+1})$$
$$= lcp(A_{Pos[p]}, A_{Pos[p+1]}) - 1 \quad (\because ③)$$
$$= height[p+1] - 1 = height[Rank[i]+1] - 1$$

□

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

3.3 알고리즘 및 분석

2절에서 살펴본 lcp 정보의 성질을 이용하면 다음과 같은 알고리즘으로 height[2..n]를 구할 수 있다.

```
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
```

정리 2. GetHeight의 시간복잡도는 $O(n)$이다.

증명. 알고리즘의 실행 시간은 루프의 가장 안쪽에 있는 6번 문장의 실행 횟수에 비례한다. 6번 문장에서는 cp의 값이 하나씩 증가하는데, 이는 5번 문장의 조건에 의해 cp < n일 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 6번 문장은 2n번 이상 수행될 수 없다. 그러므로 GetHeight의 시간복잡도는 $O(n)$이다. □

height[2..n]를 알고 있으면 접미사배열을 이용한 검색에 필요한 모든 lcp 정보를 $O(n)$시간 안에 구할 수 있다. 따라서 접미사배열과 텍스트가 주어지면 lcp정보를 $O(n)$시간안에 구할 수 있다.

4. 결론

본 논문에서는 접미사배열과 텍스트가 주어졌을 때 $O(n)$시간만에 lcp정보를 구하는 알고리즘을 제시하였다. 이를 이용하면 블록정렬방식으로 압축된 텍스트에서 접미사배열과 lcp정보를 효율적으로 생성하는 것이 가능하다.

5. 참고 문헌

[1] U.Manber and G.Myers, Suffix arrays : a new method for on-line string searches, SIAM J.Comput. (1993), 935-948

[2] K.Sadakane and H.Imai, A Cooperative Distributed Text Database Management Method Unifying Search and Compression Based on the Burrows-Wheeler Transformation, In Proc. of NewDB'98

[3] Michael Burrows and David J. Wheeler, A Block-sorting Lossless Data Compression Algorithm, Digital Systems Research Center Research Report 124, May 1994

[4] K.Sadakane, A Modified Burrows -Wheeler Transformation for Case-insensitive Search with Application to Suffix Array Compression, Proc. of Data Compression Conference '99

[5] Peter Fenwick, Block Sorting Text Compression, Proceedings of the 19th Australasian Computer Science Conference

[6] 이시은, 박근수, Suffix Array를 구축하는 새로운 알고리즘, 정보과학회논문지(A) 제24권 제7호(1997 .7)

[7] Julian Seward, http://sources.redhat.com/bzip2/