차수 3인 트리에서 가장 긴 비음수 경로를 찾는 알고리즘

(Algorithm for Finding a Longest Non-negative Path in a Tree of Degree 3)

김 성 권 †
(Sung Kwon Kim)

요 약 각 에지에 무게(양수, 음수, 0 가능)가 주어진 트리에서, 경로의 에지들의 무게의 합이 비음수이면서 길이가 가장 긴 경로를 구하는 문제를 해결하고자 한다. 차수가 3인 트리에서 가장 긴 비음수 경로를 찾는 $O(n\log n)$ 시간 알고리즘을 제시한다. n은 트리가 가지는 노드의 수이다.
키워드 : 비음수 경로, 트리

Abstract In an edge-weighted(positive, negative, or zero weights are possible) tree, we want to solve the problem of finding a longest path such that the sum of the weights of the edges in the path is non-negative. We present an algorithm to find a longest non-negative path of a degree 3 tree in $O(n\log n)$ time, where n is the number of nodes in the tree.
Key words : trees, non-negative paths

1. 서 론

배열 $A=(a_1,\ldots,a_n)$는 실수를 원소로 가지는 배열이다. 부분 배열 $(a_i,\ldots,a_j)$에 대해서 $(1\le i\le j\le n)$, 이것의 길이는 $j-i+1$이고, 합은 $a_i+\cdots+a_j$ 이며, 평균은 $\frac{a_i+\cdots+a_j}{j-i+1}$이다. 임의의 실수 θ가 주어질 때, A의 부분 배열 중에서 평균이 θ 이상이면서 길이가 가장 긴 부분 배열을 찾는 일은 생물정보학 분야에서 중요한 응용을 가지고 있다[1,5]. DNA 시퀀스나 단백질 시퀀스를 문제 성격에 따라 적당한 실수 배열로 바꿔서 앞에서 말한 부분 배열을 찾는 것이다.

그런데, 이 문제를 다음처럼 변환하면 쉽게 설명할 수 있다. 평균 $\frac{a_i+\cdots+a_j}{j-i+1}\ge\theta$이어야 하므로, 이 식을 고치면 $(a_i-\theta)+\cdots+(a_j-\theta)\ge 0$가 된다. 따라서 A와 θ로부터 배열 $A'=(a_1-\theta,\ldots,a_n-\theta)$를 구한 후, A'에서 합이 0 이상, 즉, 비음수(non-negative)이면서 길이가 가장 긴 부분 배열을 찾으면 그것이 위에서 설명한 평균이 θ 이상이면서 가장 긴 A의 부분배열이 된다. 이 일은 [1,5]에 있는 알고리즘에 의해서 $O(n)$ 시간에 가능하다.

이를 일반화하여, 트리에 적용할 수 있다. 트리 $T=(V,E)$는 노드 집합 V와 에지 집합 E로 이뤄진다. 트리의 각 에지 $e\in E$에 무게(weight)라 부르는 실수(양수, 음수, 0 모두 가능) $w(e)$가 주어진다. 트리에는 두 개의 다른 노드를 잇는 경로 P가 유일하게 존재한다. 이 때 P에 들어 있는 에지의 개수를 이 경로의 길이라 부른다. P상의 에지들의 무게의 합을 모두 합한 것을 이 경로의 무게라 하고, $w(P)$로 표시한다. 즉, $w(P)=\sum_{e\in P}w(e)$가 된다. $w(P)\ge 0$이면 P는 비음수 경로라 부른다. 트리에서 가장 긴 비음수 경로를 찾는 것은 $O(n\log^2 n)$ 시간에 가능하다[6]. 여기서 $n=|T|$이고, $|T|$는 T가 가지는 노드의 수를 의미한다.

트리에서 노드의 차수는 그 노드에 연결된 에지의 수를 말하며, 트리의 차수는 모든 노드의 차수 중에서 최대를 의미한다. [6]의 알고리즘은 트리의 차수와 무관하게 $O(n\log^2 n)$ 시간에 수행되며, 트리의 차수가 상수인 경우도 수행 시간은 변함없다. 따라서 트리의 차수가 상수인 경우, 수행 시간을 줄일 수 있는가가 관심인데, 본 논문에서는 수행 시간을 $O(n\log n)$으로 줄일 수 있음을

· 본 논문은 2003년도 중앙대학교 학술연구비 지원을 받아 이뤄짐
† 종신회원 : 중앙대학교 컴퓨터공학과 교수
skkim@cau.ac.kr
논문접수 : 2004년 1월 28일
심사완료 : 2004년 4월 9일

보인다.

본 논문에서 제시하는 방법은 (약간의 수정만 하면) 차수가 3이상의 어떤 상수인 경우도 성립하지만, 설명의 편의상 트리의 차수가 3인 경우로 한정한다. 즉, 각 노드가 많아야 세 개의 에지에 연결되며, 따라서 많아야 세 개의 다른 노드들과 이웃한다. 이런 트리는 차수가 1인 임의의 노드를 루트로 하여 rooted 트리를 만들면 이진 트리 모양을 하게 된다.

본 논문에서는 차수가 3인 트리에서 길이가 가장 긴 비음수 경로(longest non-negative path)를 찾는 것이 목적이며, 이를 위해 $O(n \log n)$ 시간 알고리즘을 제시한다. 2절에서는 알고리즘의 기본 구조를 설명하고, 3절에서 자세한 설명을 통하여 알고리즘을 제시하며 수행시간이 $O(n \log n)$임을 보인다.

2. 알고리즘의 기본 구조

본 논문의 알고리즘은 분할정복(divide-and-conquer)에 의한 재귀적(recursive) 구조를 가지고 있다. 알고리즘을 설명하기에 앞서 중요한 정의가 하나 필요하다.

차수가 3인 트리 T에서 노드 v와 그에 연결된 에지들을 삭제하면 T는 (최대) 세 개의 서브트리로 T_1, T_2, T_3 나눈다. v의 차수가 d이면 d 개의 서브트리가 남는다, $d=0,1,2,3$. 이 때, 모든 $i=1,2,3$에 대해서 $|T_i| \le |T|/2$이면 v를 **센트로이드**라 부른다. 모든 트리는 한 개 또는 두 개의 센트로이드를 가진다. 두 개를 가지는 경우는 이 둘은 반드시 이웃한다[4]. 트리가 두개의 센트로이드를 가지는 경우는 둘 중 임의의 하나를 택하면 된다.

T의 센트로이드는 $O(|T|)$ 시간에 구할 수 있다[2,3]. 가장 잘 알려진 방법은 T를 차수 1인 임의의 노드를 루트로 하여 이진 트리로 바꾼다. 노드 v에 대해서 $T(v)$를 v와 v의 모든 후손 노드들로 구성된 서브트리라 한다. 만약 v가 리프노드이면, $|T(v)|=1$이고, v가 두개의 자식 노드 u, w를 가지면, $|T(v)|=|T(u)|+|T(w)|+1$이다. 이 이진 트리를 포스트오더로 방문하면서 각 노드 v마다 $|T(v)|$를 계산한다. 이 과정 중에 처음으로 $|T(v)| \ge |T|/2$가 되는 노드 v가 그 트리의 센트로이드가 된다.

제시하려는 알고리즘은 앞에서 언급했듯이 분할정복 방법으로 수행한다.

입력: 차수가 3인 트리 T.

출력: T의 가장 긴 비음수 경로의 길이(아래 알고리즘을 조금 고치면 경로 자체를 구할 수 있으므로, 편의상 길이만 구하는 것으로 한다.)

[분할] T가 노드 하나만 가지고 있으면, 0을 반환한다. 그렇지 않으면, T의 센트로이드 c를 구한다. c와 여기에 연결된 에지들을 삭제하여 T를 (최대) 세 개의 서브트리로 T_1, T_2, T_3 나눈다.

[정복] T_1에서 가장 긴 비음수 경로의 길이를 재귀적으로 구한다. 이렇게 얻어진 경로의 길이를 L_1라 한다. 비슷하게 T_2, T_3에 대해서 재귀적으로 L_2, L_3를 구한다.

[통합] 이제, T에서 센트로이드 c를 지나면서 가장 긴 비음수 경로의 길이 L_c를 구한다. $\max\{L_1, L_2, L_3, L_c\}$를 구하면, T의 가장 긴 비음수 경로의 길이가 된다.

그림 1에서 보듯이 **[정복]** 단계에서 구한 L_1은 서브트리 T_1내에서의 가장 긴 비음수 경로의 길이다. 따라서 여기서 고려한 경로들은 c를 포함하지 않는다. 비슷하게 L_2, L_3에 대해서도 성립한다. 따라서 **[통합]** 단계에서 c를 포함하면서 가장 긴 비음수 경로의 길이 L_c를 구하여, L_1, L_2, L_3, L_c중에서 최대를 구하면 그것이 T에서의 가장 긴 비음수 경로의 길이가 되는 것이다. L_1, L_2, L_3는 재귀적으로 얻어지므로, 앞으로 설명할 것은 L_c를 구하는 효율적인 방법이다.

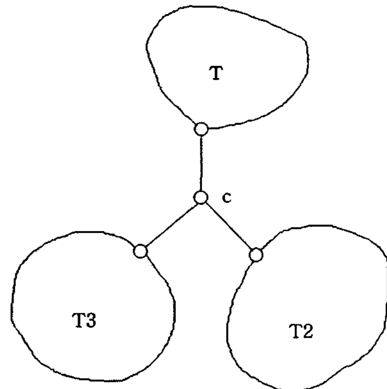

그림 1 센트로이드 c에 의해서 T_1, T_2, T_3로 나뉜 상태

$i=1,2,3$에 대해서 $n_i=|T_i|$라 한다. $W(n)$을 n개의 노드로 구성된 트리에서 우리의 알고리즘을 이용하여 가장 긴 비음수 경로 찾는 최악의 경우 수행 시간이라 두면, $W(1)=O(1)$이고, $n \ge 2$에 대해서,

$$W(n) = W(n_1) + W(n_2) + W(n_3) + c_1 n + M(n) \quad (1)$$

이 성립한다. 단, $n = n_1 + n_2 + n_3 + 1$이고, 센트로이드 성질 때문에 $n_1, n_2, n_3 \le n/2$이다. 이 식에서 $W(n_i)$는 재귀적으로 각 T_i의 해 L_i를 구하는 최악의 경우 시간이고, $c_1 n$은 [분할] 단계에서 사용한 시간이고(c_1은 상수), $M(n)$은 [통합] 단계에서 L_c를 구하는데 필요한 시

간을 n에 대한 식으로 표시한 것이다. 앞으로 $M(n)=O(n)$이라는 것을 보이면, 식 (1)은 $W(n)=O(n\log n)$이 된다.

c를 포함하는 경로들을 두 가지, 즉, c를 끝점으로 가지는 경로들과 c를 중간 노드로 가지는 경로들로 구분한다. 먼저, c를 끝점으로 가지는 경로들 중에서 가장 긴 비음수 경로의 길이를 구하기 위해서, T 전체를 c를 루트로 갖는 트리로 만든다. 루트 c부터 시작하여 프리오더 방법으로 모든 노드에 대해서 c에서 그 노드에 이르는 경로의 길이와 무게를 구한다. 그 후, 이들 중에서 비음수인 것들 만 선택하여, 가장 긴 것을 구하면 된다. 쉽게 $O(n)$ 시간에 수행할 수 있다. c를 중간 노드로 가지면서 가장 긴 비음수 경로는 찾는 방법은 본 알고리즘에서 가장 중요한 부분으로 다음 절에서 자세히 설명한다.

3. c를 중간 노드로 가지는 경로들 중에서 가장 긴 비음수 경로 찾기

c를 중간 노드로 가지는 경로는 양 끝점을 T_1, T_2, T_3에 가지게 된다. 먼저, 한쪽 끝점이 T_1에 있고 다른 쪽 끝점이 T_2에 있는 경로들 중에서 가장 긴 비음수 경로와 그 길이 $L_c^{1,2}$를 어떻게 찾는가를 설명한다. 한쪽 끝점이 T_2에 다른 쪽 끝점이 T_3에 들어 있는 경우의 $L_c^{2,3}$와, 한쪽 끝점이 T_1에 다른 쪽 끝점이 T_3에 들어 있는 경우의 $L_c^{1,3}$는 비슷한 방법으로 구할 수 있다. 그러면 $L_c=\max\{L_c^{1,2}, L_c^{2,3}, L_c^{1,3}\}$를 계산할 수 있다.

T_1의 모든 노드에 대해서 c에서 그 노드에 이르는 경로들의 길이와 무게를 모두 계산한다. 이들 중에서 길이가 i인 것들만을 모은 후, 이중에서 무게가 가장 큰 것의 값을 $A[i]$라 한다. 즉, $A[i]$는 T_1에서 c부터 길이 i만큼에 (즉, i개의 에지를 사용하여) 갈 수 있는 가장 무거운 경로의 무게이다, $1\le i\le n_1$. i가 크면, 경우에 따라 i만큼에 갈 수 있는 경로가 T_1에 없을 수 있는데, 이 경우는 $A[i]=-\infty$이다. $A[1..n_1]$을 모두 계산하기 위해서, 먼저 모든 i에 대해서, $A[i]=-\infty$로 한다. 다음, T_1의 노드들을 프리오더 순으로 방문하면서 c에서 현재 노드 v에 이르는 경로의 길이 l_v와 무게 w_v를 구한다. v의 부모노드 u의 l_u와 w_u를 알면 $l_v=l_w+1$와 $w_v=w_u+w((u,v))$를 구할 수 있다. $A[l_v]=\max(A[l_v], w_v)$를 수행하여 $A[l_v]$를 갱신한다. 이렇게 하면 $O(n_1)$ 시간에 $A[1..n_1]$을 모두 계산할 수 있다. 비슷하게, $B[j]$는 T_2에서 c부터 길이 j만큼에 갈 수 있는 가장 무거운 경로의 무게이다, $1\le j\le n_2$. $B[1..n_2]$를 모두 계산하려면 $O(n_2)$ 시간이면 된다.

이제, $A[1..n_1]$과 $B[1..n_2]$을 이용하여 $L_c^{1,2}$를 계산할 수 있다. 아래 소정리는 알고리즘을 만드는데 중요한 역할을 하는데, A와 B의 성질을 이용하여 쉽게 증명할 수 있다.

소정리 1: $L_c^{1,2}=\max(\{i+j \mid A[i]+B[j]\ge 0\}\cup\{0\})$.

즉, $L_c^{1,2}$를 구하려면 $A[i]+B[j]\ge 0$이면서 $i+j$가 최대가 되는 i,j쌍을 찾으면 된다. 만약 그런 쌍이 없으면 0이다. i,j 쌍을 모두 고려하게 되면 $O(n_1\cdot n_2)$ 시간이 걸리므로, 더 효율적인 방법을 개발해야 한다.

$j=1,\ldots,n_2$에 대해서 $i[j]$를 아래처럼 정의 한다.

- $A[i[j]]+B[j]\ge 0$ 이고, 모든 $i[j]+1\le i\le n_1$에 대해서 $A[i]+B[j]<0$ 이다.
- 모든 $1\le i\le n_1$에 대해서 $A[i]+B[j]<0$ 이면, $i[j]=-\infty$이다.

즉, $i[j]$는 j를 고정한 상태에서 $A[i]+B[j]$가 비음수가 되는 가장 큰 인덱스 i를 나타낸다. 그러면,소정리 1은 $L_c^{1,2}=\max(\{i[j]+j \mid 1\le j\le n_2\}\cup\{0\})$로 다시 쓸 수 있다. 따라서 모든 j에 대해서 $i[j]$를 어떻게 효율적으로 계산할 수 있느냐가 문제이다.

소정리 2: 임의의 $1\le j,j'\le n_2$에 대해서, $j<j'$이고 $B[j]\le B[j']$이면, 당연히 $i[j]+j<i[j']+j'$가 된다.

위 소정리에서 인덱스 j'이 인덱스 j를 **지배한다**고 말하며, 지배당하는 인덱스들은 $L_c^{1,2}$를 계산하는데 고려할 필요가 없다.

소정리 2를 이용하기 위해 $j_0, j_1, \ldots, j_m$을 다음처럼 정의한다. m은 다음에 정의 된다.

- $j_0=0$.
- $k=1,2,\ldots$에 대해서, j_k는 $\max\{B[j] \mid j_{k-1}+1\le j\le n_2\}$가 되는 $B[j]$의 인덱스이다. 최대가 되는 것이 여럿이면, 인덱스가 가장 큰 것을 j_k로 한다.
- 위와 같이 계속해서, $j_k=n_2$가 될 때의 k가 m이다.

즉, $B[j_1]$는 $B[1],\ldots,B[n_2]$에서 최대이고, $B[j_2]$는 $B[j_1+1],\ldots,B[n_2]$에서 최대이다. 이런 식으로 계속해서, $B[j_m]$는 $B[j_{m-1}+1],\ldots,B[n_2]$에서 최대이며, 이때, $j_m=n_2$ 이다. $B[j_1]>\cdots>B[j_m]$가 성립한다.

$j_0, j_1, \ldots, j_m$는 그림 2의 알고리즘 A을 수행하여 구할 수 있다. $B[0]$에 아주 큰 값을 넣고, 알고리즘을 수행하면, 마지막에 $j=0$을 출력하고 멈춘다. 출력된 j들의 개수가 $m+1$이 되고 출력 역순으로 $j_0, j_1, \ldots j_m$에 해당한다. 이를 확인하는 것은 어렵지 않다.

```
알고리즘 A

B[0]=∞;
j=k=n_2;
output (j);
while (j>0)
{
    while (B[j]≤B[k]) j--;
    output (j);
    k=j;
}
```

그림 2 알고리즘 A

소정리 3: $k=1,2,\ldots,m$에 대해서, j_k는 $j_{k-1}+1,\ldots,j_k-1$을 모두 지배한다.

소정리 4: $L_c^{1,2}=\max\left(\{i[j_k]+j_k|1\le k\le m\}\cup\{0\}\right)$.

소정리 4를 이용하면 $L_c^{1,2}$는 그림 3의 알고리즘 B에 의해서 구할 수 있다.

```
알고리즘 B

1:  i=n_1;
2:  while (i>0 and A[i]+B[j_1]<0) i--;
3:  if(i==0) { L_c^{1,2}=0; return;}
4:  t=i+j_1;
5:  k=1;
6:  while( k<m)
    {
7:    k++;
8:    while( i>0 and A[i]+B[j_k]<0 and i+j_k>t) i--;
9:    if(i==0) { L_c^{1,2}=t; return;}
10:   if( A[i]+B[j_k]≥0) t=i+j_k;
    }
11: L_c^{1,2}=t;
    return;
```

그림 3 알고리즘 B

문장 1-3은 $i[j_1]$을 계산한다. i를 n_1부터 1씩 감소하면서 처음으로 $A[i]+B[j_1]\ge 0$가 되는 i를 찾으면 그것이 $i[j_1]$이다. 만약 그런 i가 없으면, 문장 2와 3에서 $i=0$이 되는데, 이 경우는 $i[j_1]=-\infty$이며, 따라서 $L_c^{1,2}=0$이 된다. 왜냐하면, $B[j_1]$이 $B[1],\ldots,B[n_2]$ 중에서 최대이므로, 모든 i에 대해서 $A[i]+B[j_1]<0$ 이라면, 당연히 모든 i,j에 대해서 $A[i]+B[j]<0$ 이 된다. $j_2,\ldots,j_m$을 고려하지 않아도 된다. 문장 4에서 t에 $i[j_1]+j_1$을 저장한다.

문장 6-10의 while문에서, $k=2,\ldots,m$에 대하여 $B[j_2],\ldots,B[j_m]$을 순서대로 고려하면서 t를 갱신한다. 문장 8의 while에서 조건이 모두 맞으면 i를 감소한다. 만약 $A[i]+B[j_k]\ge 0$이 되면 $i[j_k]=i$가 된다. 따라서 문장 8을 끝내고, t를 갱신한다. 그러나 만약 $i+j_k=t$가 되면 역시 while문이 끝나지만, 이 경우는 $i[j_k]$를 계산할 필요가 없다. 이를 계산하더라도 $i[j_k]+j_k\le t$가 되서, t가 증가하지 않는다. t가 증가하지 않으면 $i[j_k]$를 구할 필요 없다. $i=0$이 돼서 while이 끝나면, 알고리즘을 멈추고, 현재의 t가 $L_c^{1,2}$가 된다. 앞 문단에서 설명한 것과 비슷한 이유로 $j_{k+1},\ldots,j_m$를 고려할 필요가 없다. 만약 문장 6에서 $k=m$이 되면, 문장 11로 가서 현재의 t를 $L_c^{1,2}$로 하고 멈춘다.

유의할 점은 알고리즘 B에서 i가 n_2에서 시작하여 계속 감소한다는 것이다. 이는 다음처럼 설명할 수 있다. 예를 들어, j_1과 j_2에 대해서 $i[j_1]$을 알고 있는 상태에서 $i[j_2]$를 계산할 때, i를 바로 $i[j_1]$부터 시작해도 된다. $i[j_1]+1\le i\le n_1$의 모든 i에 대해서 $A[i]+B[j_1]<0$이었으므로, 당연히 모든 $i[j_1]+1\le i\le n_1$에 대해서 $A[i]+B[j_2]<0$이 된다. $B[j_1]>B[j_2]$이기 때문이다. 비슷한 이유로, 현재의 i와 k에서 $A[i]+B[j_k]<0$이면, 당연히 $A[i]+B[j_{k+1}]<0$가 성립하므로, $A[i]+B[j_{k+1}]$에 대해서 음수 여부를 조사할 필요가 없다.

지금까지의 설명에 이용하면 아래 정리를 증명할 수 있다.

소정리 5: $L_c^{1,2}=\max\left(\{i+j\,|\,A[i]+B[j]\ge 0\}\cup\{0\}\right)$는 $O(n_1+n_2)$ 시간에 계산할 수 있다.

증명: 정확성 증명은 지금까지의 설명으로 충분하므로, 수행 시간만 분석한다. 먼저, $A[\cdot]$와 $B[\cdot]$는 각각 $O(n_1)$과 $O(n_2)$ 시간에 계산할 수 있다. 알고리즘 A는 $O(n_2)$ 시간에 가능하다. 알고리즘 B는 $O(n_1+m)$ 시간에 가능하다. □

$L_c^{2,3}$과 $L_c^{1,3}$도 소정리 5에서 $L_c^{1,2}$를 계산한 것처럼 비슷하게 계산할 수 있다. 이것으로부터 L_c를 구한다.

소정리 6: L_c는 $O(n)$ 시간에 계산할 수 있다.

소정리 6은 2절의 식 (1)에서 $M(n)$이 $O(n)$이 됨을 보인 것이다. 따라서 $W(n)=O(n\log n)$이다.

본 논문의 결론을 아래 정리처럼 내릴 수 있다.

정리 1: n개의 노드를 가지는 차수가 3인 트리의 가

장 긴 비음수 경로는 $O(n \log n)$ 시간에 계산할 수 있다.

참 고 문 헌

[1] L. Allison, Longest biased intervals and longest non-negative sum intervals, *Bioinformatics*, vol. 19(10), pp. 1294-1295, 2003.

[2] A.J. Goldman, Optimal center location in simple networks, *Transportation Science*, vol. 5, pp. 212-221, 1971.

[3] O. Kariv, S.L. Hakimi, An algorithmic approach to network location problem. I: The p-centers, *SIAM Journal on Applied Mathematics*, vol. 37, pp. 513-538, 1979.

[4] D.E. Knuth, The Art of Programming, Vol. 1. Fundamental Algorithms, 2nd Edition, Addison-Wesley, 1973.

[5] L. Wang and Y. Xu, SEGID: Identifying interesting segments in (multiple) sequence alignments, *Bioinformatics*, vol. 19(2), pp. 297-298, 2003.

[6] B.Y. Wu, K.-M. Chao, and C.Y. Tang, An efficient algorithm for the length-constrained heaviest path problem on a tree, *Information Processing Letters*, vol. 69, pp. 63-67, 1999.

김 성 권

1981년 2월 서울대학교 계산통계학과 학사. 1983년 2월 한국과학기술원 전산학과 석사. 1990년 8월 University of Washington 전산학 박사. 1991년 3월~1996년 2월 경성대학교 전산통계학과 조교수. 1996년 3월~현재 중앙대학교 컴퓨터공학과 교수. 관심분야는 생물정보학, 계산기하학, 암호응용 및 정보보호