# 크로스 링크된 단백질 서브시퀀스를 찾는 알고리즘

(Algorithm for identifying cross-linked protein subsequences)

김 성 권 †

(Sung Kwon Kim)

**요 약** 단백질의 구조를 예측하는 과정에 사용될 수 있는 다음 문제를 고려한다. 길이가 $n$이고 원소가 모두 양수인 두 배열 $A$, $B$와 양수 $M$이 주어질 때, $A[i]+\cdots+A[j]+B[k]+\cdots+B[l]=M$이 되는 부배열 쌍 $A[i],\cdots,A[j]$, $1\le i\le j\le n$과 $B[k],\cdots,B[l]$, $1\le k\le l\le n$을 모두 찾으시오. 본 논문에서는 이 문제를 $O(n^2\log n+K)$ 시간에 $O(n)$ 메모리를 사용하여 해결하는 알고리즘을 제시한다. 단, $K$는 찾은 부배열 쌍의 수이다. 기존의 결과는 $O(n^2\log n+K\log n)$ 시간과 $O(n)$ 메모리였다.

**키워드** : 단백질 시퀀스, 크로스 링크

***Abstract*** We are considering the following problem that can be used in the prediction of the structure of proteins. Given two length $n$ arrays $A, B$ with positive numbers and a positive number $M$, find all pairs of subarrays $A[i],\cdots,A[j]$, $1\le i\le j\le n$, and $B[k],\cdots,B[l]$, $1\le k\le l\le n$, such that $A[i]+\cdots+A[j]+B[k]+\cdots+B[l]=M$. This paper presents an algorithm with $O(n^2\log n+K)$ time using $O(n)$ memory, where $K$ is the number of pairs output. The previously best known one is with $O(n^2\log n+K\log n)$ time and $O(n)$ memory.

**Key words** : cross links, protein sequences

## 1. 생물정보학적인 동기

단백질은 생명체에서 중요한 역할을 하는 거대 분자이다. 단백질 분자를 풀어보면 한 개 이상의 긴 끈 형태의 펩타이드(peptide) 체인으로 이뤄져 있다는 것을 알 수 있다. 펩타이드는 수많은 아미노산(amino acid)들이 길게 연결된 것이라 생각할 수 있다. 아미노산의 종류는 20개인 것으로 밝혀졌다. 따라서, 단백질은 수많은 아미노산들로 연결된 펩타이드들이 뭉쳐져서 만들어지는 것으로 생각할 수 있다. 단백질의 기능(function)을 규정하는 것은 어떤 아미노산들이 그 단백질을 구성하는가가 아니라 그 단백질의 최종 구조(structure)라고 알려져 있다. 이를 단백질의 구조-기능(structure-function) 관계라 부르는데, 이를 밝히는 것이 생물정보학 분야의 가장 중요한 목표중의 하나이다.

· 본 연구는 2001년 한국학술진흥재단의 지원을 받았음(KRF-2001-041-E00265)

† 종신회원 : 중앙대학교 컴퓨터공학과 교수
skkim@cau.ac.kr

논문접수 : 2002년 1월 25일
심사완료 : 2002년 6월 5일

단백질의 구조-기능 관계를 알기 위해서는 단백질내의 펩타이드들 사이의 상호작용(특히, 아미노산 사이의 거리)을 밝히는 것이 필요하다. 펩타이드는 간단히 생각하면 알파벳의 크기가 20인(아미노산이 20 종류 있으므로) 스트링이라 할 수 있다. 두 펩타이드 사이의 상호작용 관계를 밝히는데 사용하는 화학적인 기법 중 하나가 크로스 링킹 (cross-linking)이다. 단백질의 펩타이드들을 화학적으로 크로스 링크를 한 후, 이 크로스 링크에 대한 정보를 얻어 분석함으로써 두 펩타이드 상호간의 관계를 알 수 있다. 크로스 링커는 시약으로서 양쪽 끝에 기능 부분이 있고 이를 연결해 주는 크로스 브릿지(bridge)로 구성된다. 양쪽 기능 부분에 따라서 서로 연결하는 아미노산이 달라진다. 길이가 가변적이고 화학적으로 절단 가능한(cleavable) 크로스 브릿지를 가지는 크로스 링커가 바람직하다.

특정 단백질을 구성하는 두 개의 펩타이드가 있을 경우, 여기에 크로스 링킹을 하면 특정 아미노산들 사이에 링크가 생긴다. (그림 1 참조) 이 아미노산들의 위치를 알아내는 일이 실험적으로 쉽지가 않다. 이를 위해 링크

된 두 펩타이드를 화학적 방법을 통해 조각들로 절단한다. 절단에 사용하는 시약에 따라서 절단되는 위치가 정해진다. 예를 들면, 단백질 소화효소인 트립신 (trypsin)은 아미노산 라이신 (lysine, 약어로 K) 바로 다음과 아미노산 아르지닌 (arginine, 약어로 R) 바로 다음을 자른다. 질량 분광계 (mass spectrometry)를 이용하여 링크가 달린 덩어리 (이것은 링크와 두 조각으로 구성됨)의 질량 $M$을 알 수 있다.

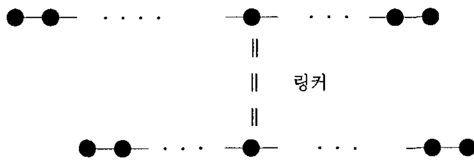


그림 1 크로스 링크된 두 펩타이드

질량 $M$을 알면 이제는 컴퓨터를 이용한(즉, 인 실리코, in silico) 방법으로 링크된 아미노산의 위치를 파악할 수 있다. 본 논문에서 다루고자 하는 문제가 이것이다. 두 펩타이드에 해당하는 아미노산 시퀀스(알파벳이 20인 스트링)가 각각 주어진다. 각 시퀀스를 읽어 가면서 라이신(K)이나 아르지닌 (R)이 나오면(앞서 말한 트립신을 사용한 것으로 가정한 경우) 서브시퀀스로 나눈다. 각 서브시퀀스에 속한 아미노산들의 질량을 더해서 서브시퀀스의 질량을 구한다. (각 아미노산의 질량은 이미 알려져 있다.) 서브시퀀스들의 질량을 배열에 저장한다. 펩타이드가 둘이므로 두 개의 배열 $A, B$가 생긴다. 이제 $A[i]+\cdots+A[j]+B[k]+\cdots+B[l]+$(링커의 질량)$=M$이 되는 모든 부배열 쌍 $A[i],\cdots,A[j]$와 $B[k],\cdots,B[l]$을 찾으면 (이것이 1단계 문제), 이게 앞에서 말한 링크가 달린 덩어리의 후보가 된다. 물론, 후보가 많이 생길 수도 있다. $A[i]+B[k]+$(링커의 질량)$=M$인 원소 쌍을 찾지 않고 부배열 쌍을 찾는 이유는 화학적으로 절단할 때, 두 개의 K 또는 R이 아주 가까이 있으면 앞쪽에서는 절단이 일어나지 않을 수도 있고, 주변의 화학적 환경 때문에 어떤 K나 R에서는 절단이 불가능할 수도 있기 때문이다. $A$의 서브시퀀스와 $B$의 서브시퀀스 중에서 어느 아미노산 사이에 링크가 연결되었는지를 구하는 것은 또 다른 문제이다. (이것이 2단계 문제)

논문 [1]에는 1, 2 단계 문제 각각에 대한 알고리즘을 제시하고 있다. 본 논문에서 1단계 문제만 다루며, 논문 [1]에서 제시한 알고리즘 보다 더 효율적인 방법을 제시한다. 2단계 문제에 대한 알고리즘은 논문 [1]의 것을 사용한다.

## 2. 문제 정의와 간단한 해법

여기서는 앞 절에서 설명한 생물정보학 문제를 정확하게 정의하여 컴퓨터학 문제로 전환한다. 먼저, 길이가 $n$인 두 배열 $A$와 $B$에 대해서, 부배열의 합을 $A_{i,j}=A[i]+\cdots+A[j]$ 와 $B_{i,j}=B[i]+\cdots+B[j]$ $(1\le i\le j\le n)$로 표기하자. 꼭 두 배열의 길이가 같을 필요는 없지만 표기를 간단히 하기 위해 그렇게 가정한다. 그러면, 우리가 해결하고자 하는 문제는 다음처럼 정의된다.

**부배열 합 문제 (subarray sum problem, SSP)**: 길이가 $n$이고 원소가 모두 양수인 두 배열 $A$, $B$와 양수 $M$이 주어질 때, $A_{i,j}+B_{k,l}=M$이 되는 부배열 쌍 $A[i],\cdots,A[j]$, $1\le i\le j\le n$과 $B[k],\cdots,B[l]$, $1\le k\le l\le n$을 모두 찾으시오.

크로스 링커의 질량은 상수이므로 빼고 생각해도 무방하다. 본 논문에서는 부배열 합 문제를 $O(n^2\log n+K)$ 시간에 $O(n)$ 메모리를 사용하여 해결할 수 있다는 것을 증명하고자 한다. 단, $K$는 찾은 부배열 쌍의 수로 출력의 크기이다.

$N=\frac{n(n+1)}{2}$라 놓자. $A$와 $B$ 각각에서 나올 수 있는 부배열의 수가 $N$이므로, 단순하게 생각하면 $N^2$개의 부배열 쌍을 모두 고려해서 무게의 합이 $M$이 되는 쌍을 모두 찾으면 된다. 이 경우 걸리는 시간이 $\Omega(n^4)$이 된다.

SSP를 해결하는 효율적인 다른 방법을 찾기 위해서 간단한 다음 문제를 먼저 해결하자.

**쌍 합 문제 (pair sum problem, PSP)**: 두 배열 $D_A$와 $D_B$에 양수들이 있다. 양수 $M$이 주어질 때, $D_A[i]+D_B[j]=M$이 되는 $i, j$ 쌍을 모두 구하시오.

먼저, $D_A$의 양수들이 모두 다르고, $D_B$의 양수들이 모두 다르다고 가정하자. 이 가정이 성립하지 않는 경우는 나중에 설명한다. $D_A$의 원소들은 큰 것부터 차례대로 하나씩 고려하고 $D_B$의 원소들은 작은 것부터 차례대로 하나씩 고려한다. 먼저, $D_A$의 수 중에서 $M$ 이상인 것은 모두 삭제하고, 그림 2를 수행한다. max($X$)와 min($X$)는 $X$의 최대와 최소를 찾는 명령이고, delete $(y, X)$는 $y$를 $X$에서 삭제하는 명령이다.

알고리즘을 설명하면, $D_A$가 공집합이면 (1)에서 알고리즘을 멈추고, 아니면 (2)에서 $D_A$의 최대 ($v$)를 찾는다. (3)-(4)에서는 $D_B$에 아직까지 고려하지 않은 원소가 남아 있으면 그 중에서 최소($w$)를 찾는다. (5)에서

```
(1) while( D_A≠∅ ) {
(2)     v=max( D_A );
(3)     while( D_B≠∅ ) {
(4)         w=min( D_B );
(5)         if v+w<M then delete( w, D_B );
            else break;
        }
(6)     if D_B=∅ then break;
(7)     if v+w=M then
            output i,j s.t. v=D_A[i] and w=D_B[j];
(8)     delete( v, D_A );
    }
```

그림 2 중복이 없는 경우 PSP를 해결하는 알고리즘

```
(1) while( D_A≠∅ ) {
(2)     v=max( D_A );
(3)     while( D_B≠∅ ) {
(4)         w=min( D_B );
(5)         if v+w<M then delete( w, D_B );
            else break;
        }
(6)     if D_B=∅ then break;
(7)     if v+w=M then
            output all pairs i,j s.t. i∈D_{A,v} and j∈D_{B,w};
(8)     delete( v, D_A );
    }
```

그림 3 중복이 있는 경우 PSP를 해결하는 알고리즘

$v+w<M$이면 현재의 $w$는 버리고 다음으로 작은 원소 ($w$)를 찾기 위해서 (3)으로 간다. (3)-(5)를 반복 수행하다가 $v+w\geq M$가 되면 while 루프를 멈춘다. (6)에서 while 루프를 나온 이유가 $D_B=\emptyset$ 인가를 검사하여 맞으면 알고리즘을 멈추고, 아니면 (7)로 와서 $v+w=M$이 맞는가를 조사한다. 맞으면 $i,j$ 쌍을 출력한다. (8)에서 ($v+w\geq M$) $v$를 삭제하고 (1)로 간다.

전체적인 구조를 보면, $D_A$의 원소들을 큰 것부터 고려하는 while 루프가 있고, 그 안에 $D_B$의 원소들을 작은 것부터 고려하는 while 루프가 있다. $D_A$와 $D_B$의 원소들이 모두 다르다는 사실을 이용하면 이 알고리즘의 정확성을 증명하는 것은 어렵지 않다.

$D_A$와 $D_B$는 소트된 배열을 이용하여 구현하면 된다. $m=|D_A|+|D_B|$라 두면 $D_A$와 $D_B$를 소트하는 시간은 $O(m\log m)$이다. $D_A$의 최대 원소를 가리키는 인덱스 ($i$)와 $D_B$의 최소 원소를 가리키는 인덱스 ($j$)를 두면 max, min , delete 문장 모두 수행 시간은 $O(1)$이다. 다른 모든 문장 역시 $O(1)$에 가능하므로, 수행 시간은 $O(m+K)$이다. 여기서 $K$는 출력되는 쌍의 수이다. 따라서 전체 수행 시간은 $O(m\log m+K)$이다.

지금까지는 $D_A$의 양수들이 모두 다르고, $D_B$의 양수들도 모두 다르다고 가정했는데, 이제 이 가정이 성립하지 않고 $D_A$나 $D_B$에 같은 수들이 중복으로 있는 경우를 살펴보자. 먼저, $v\in D_A$에 대해서 $D_{A,v}=\{i|v=D_A[i]\}$, $w\in D_B$에 대해서 $D_{B,w}=\{i|w=D_B[i]\}$라 정의하자. 즉, 같은 값을 가지는 인덱스들을 각각 모아서 집합을 만든다.

$D_A$의 수 중에서 $M$ 이상인 것은 모두 삭제한다. 두 배열 $D_A$와 $D_B$를 소트하여 $D_{A,v}$들과 $D_{B,w}$들을 구하여 따로 저장한다. 또, $D_A$에서 중복된 원소를 모두 삭제하여 서로 다른 원소들만 $D_A$에 남긴다. 비슷하게 서로 다른 원소들만 $D_B$에 남긴다. 그리고 그림 3의 알고리즘을 수행한다.

이 알고리즘은 근본적으로 그림 2의 알고리즘과 같으므로 그 정확성은 쉽게 증명 가능하다. 수행 시간은 $D_A$와 $D_B$를 소트해서 $D_{A,v}$들과 $D_{B,w}$들을 모두 구하는 시간은 $O(m\log m)$이다. 알고리즘에서 (7)번 문장을 제외한 다른 것은 앞서와 동일하다. (7)에서 $i,j$ 한 쌍을 출력하는 시간은 $O(1)$이므로, 알고리즘 수행 시간은 $O(m+K)$이다. 전체적으로, 소트 시간까지 포함하여 계산하면 수행 시간은 $O(m\log m+K)$이다. 사용하는 메모리 양은 $O(m)$이다. 지금까지는 PSP 문제에 대한 알고리즘 설명이었다.

이제 원래 문제 SSP로 돌아가서 이를 해결하는 방법을 살펴보자. $D_A$에 $A$가 만드는 $N$개의 부배열의 합들($A_{i,j}$)과 그것의 시작/끝 인덱스 $(i,j)$을 저장하고, $D_B$에 $B$가 만드는 $N$개의 부배열의 합들($B_{k,l}$)과 그것의 시작/끝 인덱스 $(k,l)$을 역시 저장한다. 그런 다음 그림 3 알고리즘을 적용하면 된다. 물론, (7)에서 출력할 때는 해당하는 두쌍의 시작/끝 인덱스 $i,j,k,l$을 출력한다. 수행시간은 $O(n^2\log n+K)$이고 사용하는 메모리 양은 $O(n^2)$이다.

## 3. 메모리 양을 줄이는 방법

앞 절의 알고리즘을 향상 시켜서 수행시간을 줄이는 것은 매우 어려워 보인다. 어쩌면 $\Omega(n^2\log n+K)$가 하한일지도 모른다. 따라서 메모리를 줄이는 것이 다음 목표이다.

메모리를 줄이는 방법의 아이디어를 설명하기 위해서 $D_A$를 예로 든다. 특정 인덱스 $i$에 대해서, 만약 $A_{i,j}$가 현재 $D_A$안에 있으면 ($i<j$), $A_{i,j-1}$은 그림 3의 (2)번 문장 $v=\max(D_A)$에서 절대로 $v$가 될 수 없다. $A_{i,j}>A_{i,j-1}$이기 때문이다. 이는 다른 모든 $i$에 대해서도 마찬가지로 성립한다. 따라서, $D_A$에는 각 인덱스 $i$를 시작 인덱스로 하는 부배열의 합 중에서 가장 큰 것

하나만 있으면 된다. 만약 $A_{i,j}$가 $D_A$에서 삭제되면 $A_{i,j-1}$을 대신 추가시킨다. 비슷하게 $D_B$에 대해서도 이것이 성립하는데, 차이점은 $D_B$는 작은 것부터 차례로 고려해야 하므로 만약 $B_{k,l}$이 삭제되면 대신 $B_{k,l+1}$을 추가시킨다는 것이다.

지금까지는 $D_A$와 $D_B$를 배열로 생각했으나, 잠시 구현 방법이 아직 미정인 추상적인 자료 구조로 생각한다. 이들의 구현 방법은 나중에 자세히 설명한다. 초기의 $D_A$를 구성하기 위해서, 먼저 모든 $1 \le i \le n$에 대해서 $A_{i,j(i)} < M \le A_{i,j(i)+1}$를 만족하는 인덱스 $j(i)$를 구한다. $A_{i,j} \ge M$인 부배열은 고려할 필요가 없다. 초기에는 $D_A = (A_{1,j(1)}, \ldots, A_{n,j(n)})$이다. $D_A$에서 동일한 값을 가지는 시작/끝 인덱스들의 집합을 모두 구한다. 즉, $v \in D_A$에 대해서 $D_{A,v} = \{(i, j(i)) \mid v = A_{i,j(i)}\}$를 구한다. 초기 $D_B$를 구성하는 일은 더 간단하다. 초기에는 $D_B = (B_{1,1}, \ldots, B_{n,n})$이다. 역시 $w \in D_B$에 대해서 $D_{B,w} = \{(k,k) \mid w = B_{k,k}\}$도 구한다. 이제 그림 4의 알고리즘을 수행한다.

```
(1) while( D_A≠∅ ) {
(2)   v=max( D_A );
(3)   while( D_B≠∅ ) {
(4)     w=min( D_B );
(5)     if  v+w<M  then {
            delete( B_{k,l}, D_B ) for each  (k,l)∈D_{B,w};
            add( B_{k,l+1}, D_B ) for each  (k,l)∈D_{B,w}
                 unless  B_{k,l+1}≥M  or  n<l+1;
        }
        else break;
    }
(6) if  D_B=∅  then break;
(7) if  v+w = M then output all  i,j,k,l
                s.t.  (i,j)∈D_{A,v}  and  (k,l)∈D_{B,w};

(8) delete( A_{i,j}, D_A ) for each  (i,j)∈D_{A,v};
    add( A_{i,j-1}, D_A ) for each  (i,j)∈D_{A,v}  unless
      i>j-1;
  }
```

그림 4 적은 메모리를 사용하는 알고리즘

그림 3의 알고리즘과 다른 점은 (5)에서 $w = B_{k,l}$인 원소들을 모두 $D_B$에서 삭제하고, 대신 $B_{k,l+1}$를 추가시킨다. add( $y, X$)는 $y$를 $X$에 추가하는 명령이다. 유사하게 (8)에서 $v = A_{i,j}$들을 모두 삭제하고 대신 $A_{i,j-1}$들을 추가한다.

$D_A$와 $D_B$ 모두 $n$개 이하의 원소를 가진다는 것은 자명하다. 문제는 $D_A$와 $D_B$를 어떻게 만들고, 삭제와 추가 연산을 어떻게 구현하느냐가 문제이다. 가장 쉽게 생각할 수 있는 것이 $D_A$를 max-heap으로 $D_B$를 min-heap으로 구현하는 것이다. 앞에서도 언급했지만 (2)에서 $v = \max(D_A)$이므로 (8)에서 $D_A$로 부터 삭제되는 원소들을 항상 $D_A$에서 최대 원소이다. 최대를 찾고 이를 삭제하는 것은 max-heap의 기본 연산에 해당한다. 또한, $D_{A,v}$가 실제 필요한 경우 ((7)과 (8) 문장)는 $v$가 $D_A$에서 최대가 되는 경우뿐이다. 이때의 $D_{A,v}$는 $D_A$에서 최대를 찾는 일을 반복하면 구할 수 있다. $D_B$에서 최소를 찾아 이를 삭제하는 것도 min-heap의 기본 연산이다. $D_{B,w}$가 실제 필요한 경우((5)와 (7) 문장)는 $w$가 $D_B$에서 최소인 경우뿐이다. 이때는 $D_B$에서 최소를 찾는 일을 반복하면 $D_{B,w}$를 얻을 수 있다. 따라서, 초기의 heap $D_A$와 $D_B$를 구성하는데 $O(n)$ 시간이 걸리고, 다음부터 heap 연산은 삭제나 추가 모두 $O(\log n)$ 시간 걸리므로 그림 4 알고리즘 수행은 $O((n^2+K)\log n)$ 시간 걸린다. 사용하는 메모리 양은 $D_A$와 $D_B$만 저장하면 되므로 $O(n)$이다.

논문 [1]에서처럼 $D_A$와 $D_B$를 red-black tree [2]를 이용해서 구현해도 된다. 물론, 최대, 최소, 추가, 삭제 연산을 모두 $O(\log n)$ 시간에 하는 다른 탐색 구조를 이용해도 괜찮다. 어느 것을 사용하더라도 heap으로 구현하는 것 보다 구현 과정이 더 복잡하다.

결과적으로 이 알고리즘을 그림 3과 비교해 보면 메모리 양은 $O(n^2)$에서 $O(n)$으로 줄었지만 수행 시간 중에서 답 출력시간이 차지하는 부분이 $O(K)$에서 $O(K\log n)$으로 늘었다.

## 4. 답 출력시간을 줄이는 방법

이제 메모리 양은 그대로 $O(n)$으로 유지하면서 답 출력시간을 $O(K)$으로 줄이는 방법을 생각해 보자. 답 출력시간이 $O(K\log n)$이 되는 이유는 (7)번 문장에서 답 하나를 출력할 때 이 답을 찾는 시간 $O(\log n)$이 heap이나 red-black tree에는 필요하기 때문이다. 다시 생각해 보면, $D_{A,v}$나 $D_{B,w}$를 미리 만들어 가지고 있지 않고 필요한 출력 시점에서 이를 구성하기 때문이다. 이를 극복하려면, 예를 들어 heap에서 같은 값 $v$가 여러 번 나오면 이들을 각각 따로 저장하지 말고, 하나의 노드에만 $v$를 저장하고 대신 그 노드에 $D_{A,v}$를 첨부시키면 된다. 그런데, 이것을 heap에서 구현하기는 어려우므로 다른 형태의 자료구조가 필요하다.

그런 자료구조를 $D_A$를 예로 해서 설명한다. $n$개의 리프 노드를 가지는 완전이진 트리 $T$를 구성한다. $T$의 리

프 노드들은 왼쪽부터 차례로 $x_1, \cdots, x_n$이라 표시하자. 각 $i$에 대해서 $\mathrm{val}(x_i)=A_{i,j(i)}$, $\mathrm{set}(x_i)=\{(i,j(i))\}$로 놓는다. 그리고 토너멘트 방식으로 $T$의 모든 내부 노드 $x$에 대해서 val$(x)$와 set$(x)$를 정한다. 그림 5에서 $T$의 내부 노드 $x$의 왼쪽 자식 노드를 $y$, 오른쪽 자식 노드를 $z$라 하자. val$(y)$와 val$(z)$ 중 큰 것이 val$(x)$가 되며, 큰 것의 set()가 set$(x)$가 된다. $\mathrm{val}(y)=\mathrm{val}(z)$이면 $\mathrm{set}(y)\cup\mathrm{set}(z)$가 set$(x)$가 된다. 자기의 set()가 부모 노드로 올라가면 그것은 공집합이 된다.

```
if   val(y) > val(z) then {
          val(x) = val(y);
          set(x) = set(y);
          set(y) = ∅;
}
else if  val(y) < val(z) then {
          val(x) = val(z);
          set(x) = set(z);
          set(z) = ∅;
}
else {   /*  val(y) = val(z) */
          val(x) = val(y);
          set(x) = set(y) ∪ set(z);
          set(y) = set(z) = ∅;
}
```

그림 5 val$(x)$와 set$(x)$를 계산하기

set(x)는 링크드 리스트를 이용하여 구현한다. 노드 $x$가 set(x)를 가리키는 포인터를 가지고 있다. $\mathrm{set}(x)=\mathrm{set}(y)$는 $y$의 포인터를 $x$의 포인터로 옮겨 주면 된다. $\mathrm{set}(x)=\mathrm{set}(y)\cup\mathrm{set}(z)$는 두 개의 링크드 리스트를 하나로 합친 다음 $x$가 그것을 가리키도록 하면 된다. 따라서, 그림 5의 일은 $O(1)$ 시간이 소요되므로 $T$를 만드는 시간은 $O(n)$이다. $\sum_{x\in T}|\mathrm{set}(x)|\le n$이므로, $T$와 val(x), set$(x)$들이 사용하는 메모리 총 양은 $O(n)$이다.

$T$의 루트노드 $r$의 val$(r)$이 $\max(D_A)$이고, set$(r)$이 $D_{A,\mathrm{val}(r)}$가 된다. 이제, set$(r)$의 원소들이 모두 삭제된 후, $T$를 어떻게 갱신하는가를 설명한다. 먼저, $\mathrm{set}(r)=\{(i,j)\}$로 set$(r)$이 하나의 원소로만 이뤄진 경우를 살펴보자. 이 경우, $A_{i,j}$가 삭제되고 $A_{i,j-1}$이 추가되어야 한다. $T$에서 리프 노드 $x_i$에서 루트 노드에 이르는 패쓰를 $P$라 하면, $P$상의 모든 노드 $x$에 대해서 $\mathrm{set}(x)=\emptyset$이다. $P$상의 노드 $x$의 자식노드 중에서 $P$에 속하지 않는 노드를 $z$라 하면, 그런 모든 $z$에 대해서 $\mathrm{set}(z)\ne\emptyset$이다. 따라서, $\mathrm{val}(x_i)=A_{i,j-1}$, $\mathrm{set}(x_i)=\{(i,j-1)\}$로 놓은 후, $P$의 모든 노드 $x$에 대해서 bottom-up 방식으로 그림 5의 방법을 이용하여 val$(x)$와 set$(x)$를 갱신하면 된다. $O(\log n)$ 시간 걸린다.

다음으로 $\mathrm{set}(r)=\{(i_1,j_1),(i_2,j_2),\cdots\}$로서 set$(r)$가 둘 이상의 원소를 가진 경우를 보자. $i_1<i_2<\cdots$라 가정하자. 이 가정은 그림 5에서 $\mathrm{set}(x)=\mathrm{set}(y)\cup\mathrm{set}(z)$를 수행할 때 set$(y)$의 링크드 리스트를 앞쪽에, set$(z)$의 링크드 리스트를 뒤쪽에 놓고 연결하면 성립한다. 리프 노드 $x_{i_1}$에서 루트에 이르는 패쓰를 $P_1$이라 하자. $\mathrm{val}(x_{i_1})=A_{i_1,j_1-1}$, $\mathrm{set}(x_{i_1})=\{(i_1,j_1-1)\}$로 고친다. $P_1$상의 임의의 노드 $x$의 자식노드 $y,z$ 중 $z$를 $P_1$상에 있지 않는 자식 노드라 하자. 만약 $\mathrm{set}(z)\ne\emptyset$이면 그림 5의 방법으로 val$(x)$와 set$(x)$를 갱신하고 그 위 노드로 올라간다. 그렇지 않으면, val$(z)$와 set$(z)$를 먼저 계산해야 하므로 리프 노드 $x_{i_2}$로 가서 $P_2$에 대해서 $P_1$에서 했던 일과 같은 일을 반복한다. 그래서 val$(z)$와 set$(z)$가 계산되면 다시 $P_1$으로 돌아와서 val$(x)$와 set$(x)$를 갱신한 후, $P_1$을 따라 올라간다. 이런 식으로 모든 $i_1, i_2, \cdots$에 대해서 일을 마치면 $T$에 대한 갱신이 끝난다. $T$의 높이가 $O(\log n)$이므로, 갱신 시간은 $O(|\mathrm{set}(r)|\cdot\log n)$이 된다. 이 갱신 작업을 그림 4의 알고리즘 수행 전 과정을 생각하면 $\sum|\mathrm{set}(r)|=O(n^2)$이므로, 갱신 작업에 드는 총시간은 $O(n^2\log n)$이다.

$D_B$에 대해서도 비슷하게 완전이진 트리 $U$를 만들고 모든 노드 $x$에 대해서 val$(x)$와 set$(x)$를 계산한다. 여기서, 리프 노드 $x_i$의 초기값은 $\mathrm{val}(x_i)=B_{i,i}$와 $\mathrm{set}(x_i)=\{(i,i)\}$이다. 그림 5의 방법은 큰 수가 올라가는 토너멘트이지만, $U$는 작은 수가 올라가는 토너멘트로 해야 한다. $U$의 루트를 $s$라 하자. 앞서와 비슷하게 set$(s)$에 들어 있는 모든 원소를 삭제하고 $U$를 갱신하는 작업은 역시 $O(|\mathrm{set}(s)|\cdot\log n)$ 시간에 가능하다. 역시 $\sum|\mathrm{set}(s)|=O(n^2)$이므로 갱신 작업에 드는 총시간은 $O(n^2\log n)$이다.

답 출력시간을 보면, 트리 $T$와 $U$가 만들어지면, 각각의 루트 $r$과 $s$에 set$(r)$와 set$(s)$가 계산되어 있다. 둘 다 링크드 리스트이므로 $(i,j)\in\mathrm{set}(r)$와 $(k,l)\in\mathrm{set}(s)$인 한 개의 출력 $(i,j,k,l)$에 $O(1)$ 시간이 사용된다. 따라서 답 출력시간은 $O(K)$이다.

지금까지의 설명을 종합해 보면 정리 1에 대한 증명이 완성된 것을 알 수 있다.

**정리 1.** 부배열 합 문제는 $O(n^2\log n+K)$ 시간에 $O(n)$ 메모리를 사용하여 해결할 수 있다. 단, $K$는 찾은 부배열 쌍의 수이다.

## 5. 결 론

본문에서는 부배열 합 문제를 $O(n^2 \log n + K)$ 시간에 $O(n)$ 메모리를 사용하여 해결할 수 있음을 보였다. 앞으로 연구할 점은 $O(n^2 + K)$ 시간에 가능한가이다. 가능하다면, 메모리 양을 $O(n)$으로 유지하면서 가능한지 아니면 더 많은 메모리가 필요할지가 문제이다.

## 참 고 문 헌

[1] T. Chen, J.D. Jaffe, and G.M. Church, *Algorithms for identifying cross-links via tandem mass spectrometry,* Proceedings of the Fifth Annual International Conference on Computational Biology (RECOMB 2001), pages 95--102 (2001).

[2] E. Horowitz, S. Sahni, and S. Anderson-Freed, *Fundamentals of Data Structures in C,* Computer Science Press, 1993.

김 성 권
정보과학회논문지: 시스템 및 이론
제29권 3호 참조