단백질 시퀀스와 가중치 스트링에 대한 탐색 알고리즘

(Searching Algorithms for Protein Sequences and Weighted Strings)

김 성 권 †

(Sung Kwon Kim)

요 약 단백질 시퀀스처럼 가중치를 가지는 스트링에 대한 탐색 알고리즘을 개발한다. Σ를 알파벳이라 하고 모든 $a \in \Sigma$에 대해서 무게 $\mu(a)$가 주어진다고 하자. 스트링 $A = a_1 a_2 \cdots a_n$에서 (단, 모든 $a_i \in \Sigma$), 서브스트링 $A(i,j) = a_i a_{i+1} \cdots a_j$로 정의하면, 이것의 무게는 $\mu(A(i,j)) = \mu(a_i) + \mu(a_{i+1}) + \cdots + \mu(a_j)$가 된다. 다루고자 하는 문제는 스트링 A를 사전 처리하여 탐색 자료구조를 만드는데, 이 자료구조는 나중에 질문 무게 M이 주어진 경우, $M = \mu(A(i,j))$인 서브스트링 $A(i,j)$가 있는가 라는 질문에 응답하는데 사용된다. 본 논문에서는 기존의 결과를 향상시키는 알고리즘을 제시한다. 기존의 알고리즘의 경우 $O(n)$ 만큼의 메모리를 사용하는 탐색 자료구조를 이용하여 $O(\frac{n \log \log n}{\log n})$ 시간에 질문응답을 하였으나, 본 논문의 알고리즘은 질문 응답 시간은 그대로 유지하면서 메모리만 $O(\frac{n}{\log n})$으로 줄인다.

키워드 : 단백질 시퀀스, 가중치 스트링, 탐색 알고리즘

Abstract We are developing searching algorithms for weighted strings such as protein sequences. Let Σ be an alphabet and for each $a \in \Sigma$ its weight $\mu(a)$ is given. Given a string $A = a_1 a_2 \cdots a_n$ with each $a_i \in \Sigma$, a substring $A(i,j) = a_i a_{i+1} \cdots a_j$ has weight $\mu(A(i,j)) = \mu(a_i) + \mu(a_{i+1}) + \cdots + \mu(a_j)$. The problem we are dealing with is to preprocess A to build a searching structure, and later, given a query weight M, the structure is used to answer the question of whether there is a substring $A(i,j)$ such that $M = \mu(A(i,j))$. In this paper an algorithm that improves over the previous result will be presented. The previously best known algorithm answers a query in $O(\frac{n \log \log n}{\log n})$ time using a searching structure that requires $O(n)$ amount of memory. Our algorithm reduces the memory requirement to $O(\frac{n}{\log n})$ while achieving the same query answer time.

Key words : Protein sequence, weighted string, searching algorithm

1. 생물정보학적인 동기

단백질은 모든 생명체에서 중요한 역할을 담당하는 거대 분자로서, 아미노산이라 불리는 작은 분자들이 일렬로 연결되어 만들어진다. 이런 아미노산 시퀀스를 단백질의 일차 구조라 부른다. 단백질은 적게는 100개 미만부터 많게는 수천 개의 아미노산들로 이뤄지는데, 주로 300~600개 정도로 구성된다. 인간의 몸에서 발견되는 아미노산의 종류는 모두 20개이다. 따라서 단백질의 일차 구조는 알파벳 크기가 20인 스트링으로 볼 수 있다.

이미 밝혀진 단백질에 대한 정보는 PDB [1]나 SWISS-PROT[2]와 같은 데이터 베이스에 등록되어 있다. 연구자가 새로 단백질을 추출했을 때, 이것이 이미 알려진 단백질인지를 알고 싶은 경우가 있다. 이 경우 가장 확실한 방법은 단백질 시퀀싱(sequencing)으로 그 단백질의 일차구조를 밝히는 것이다. 시퀀싱은 화학적인 방법으로, 추출한 단백질 분자를 화학 실험으로 아미노산을 하나씩 알아내서 모든 아미노산을 밝혀 단백질 시퀀스를 얻는다. 이렇게 알아낸 일차 구조(즉, 시퀀스)를 가지고 데이터베이스를 탐색하는 것이다. 그러나 단백질 시퀀싱은 시간과 노력이 많이 들어서 대용량 고속 처리를 요하는 곳에서는 바람직하지 않는 방법이다.

· 본 논문은 2001학년도 중앙대학교 학술연구비 지원에 의한 것임.

† 종신회원 : 중앙대학교 컴퓨터공학과 교수

skkim@cau.ac.kr

논문접수 : 2002년 1월 25일

심사완료 : 2002년 6월 5일

그래서 필요한 것이 시퀀싱을 하지 않고 데이터 베이스를 탐색하는 것이다. 추출한 단백질을 화학적 방법으로 서브스트링으로 나눈 후, 각 서브스트링의 질량을 잰다. 이 질량들을 모으면 그 단백질에 대한 일종의 "fingerprint"가 생긴다. 이 fingerprint를 가지고 데이터베이스를 탐색한다. 목표는 각 질량에 대응하는 서브스트링을 모두 가지고 있는 단백질 시퀀스를 데이터 베이스에서 찾는 것이다. 따라서, 데이터베이스에 있는 각 단백질 시퀀스마다 탐색 자료구조를 미리 형성하여, 주어진 질량에 대해서, 이에 대응하는 서브 스트링을 이 단백질 시퀀스가 가지고 있는가를 빨리 알 수 있도록 해야 한다.

2. 문제 정의

본 절에서는 앞에서 언급한 생물정보학 문제를 정확하게 정의하여 컴퓨터학 문제로 바꾼다. 본 논문에서는 스트링과 시퀀스, 질량과 무게를 혼용한다. 그리고 알파벳 Σ는 고정된 크기를 갖는다. 단백질 시퀀스의 경우, $|\Sigma|=20$이다. 모든 $a\in\Sigma$에 대해서 질량 $\mu(a)$가 정해져 있다. 길이 n인 주어진 단백질 시퀀스 (스트링) $A=a_1a_2\cdots a_n$에서 (단, 모든 $a_i\in\Sigma$), 서브스트링 $A(i,j)=a_ia_{i+1}\cdots a_j$로 (단, $1\le i\le j\le n$) 정의하면, 이것의 질량은 $\mu(A(i,j))=\mu(a_i)+\mu(a_{i+1})+\cdots+\mu(a_j)$가 된다.

본 논문에서 다루고자 하는 문제는 스트링 A를 사전 처리 (preprocessing) 하여 탐색 자료구조를 만드는데, 이 자료구조는 나중에 질문 (query) 질량 M이 주어진 경우, $M=\mu(A(i,j))$인 서브스트링 $A(i,j)$가 있는가 라는 질문에 응답하는데 사용된다. 여기서 알고리즘의 효율을 측정하는 척도는 탐색 자료구조 저장에 필요한 메모리 양과 질문 응답 시간이다. 사전처리는 처음 한 번 만하면 되므로 사전 처리 시간은 별도로 측정하지 않는다. 참고로 본 논문에서 제시하는 방법의 사전 처리 시간은 $O(n^2\log n)$이다.

이 문제를 해결하는 간단한 방법은 사전 처리를 전혀 하지 않고 A를 그대로 저장하는 것이다. 질문 M이 주어지면, A를 탐색하면서 쉽게 $O(n)$ 시간에 다음처럼 응답을 할 수 있다. 두 인덱스 $i\le j$에 대해서 $\mu(A(i,j))<M$이면 j를 $\mu(A(i,j))\ge M$가 될 때까지 1씩 증가시킨다. 여기서, 만약 $\mu(A(i,j))=M$이면 답을 찾았으므로 멈추면 되고, 아니면 i를 $\mu(A(i,j))\le M$가 될 때까지 1씩 증가시킨다. $\mu(A(i,j))=M$이면 답을 찾은 것이고, 아니면 $\mu(A(i,j))<M$이므로 앞의 과정을 반복하면 된다. 물론 초기치는 $i=j=1$이다. 따라서 메모리 양과 응답 시간이 모두 $O(n)$이다.

다른 방법은 모든 $i\le j$에 대해서 $\mu(A(i,j))$를 모두 계산하여, 이들을 소트하여 저장한다. 질문에 대한 응답은 소트된 리스트를 이진 탐색하면 된다. 메모리 양은 $O(n^2)$이고 응답 시간은 $O(\log n)$이다. 이 경우, 사전 처리 시간을 분석해 보면 $O(n^2\log n)$이다.

스트링이나 시퀀스에 대한 문제들은 이미 컴퓨터 분야에서 널리 알려진 것이며 생물정보학에서도 많이 응용하고 있다. 예를 들어, 두 스트링의 유사도 측정, 반복 스트링 구하기, 스트링 매칭 등이 있는데 이들은 [4]에 잘 설명되어 되어 있다. 그러나 본 논문에서 다루고자 하는 문제가 처음으로 언급된 것은 논문 [3]에서다. [3]에서는 방금 앞에서 설명한 두 가지 방법 보다 좋은 방법을 찾는 것을 목표로 하고 있다. 즉, 메모리 양이 $o(n^2)$이고 응답시간이 $o(n)$인 알고리즘이 바람직한 것으로 보고 (둘 다 소문자 o임), "LOOKUP"이라는 알고리즘을 제시하고 있는데, 사용하는 메모리 양은 $O(n)$이고 응답 시간은 $O(\frac{n\log\log n}{\log n})$이다. 또, 그 논문에서 제시한 "CLUSTER"라 부르는 알고리즘은 $O(n+BL)$ 메모리와 $O(\log L+B\log n)$ 응답시간을 갖는 것으로서, 비록 메모리 양이 $o(n^2)$이고 응답시간이 $o(n)$이라는 것은 증명하지 못 했지만, 실용적으로 좋은 효율을 갖는다고 설명하고 있다. B와 L은 나중에 설명한다.

본 논문의 주요 결과는 응답 시간은 LOOKUP과 같이 $O(\frac{n\log\log n}{\log n})$로 유지하면서 메모리 양을 $O(\frac{n}{\log n})$으로 줄인 알고리즘을 개발한 것이다. 이를 얻기 위해서 CLUSTER를 향상시킨 CLUSTER2 알고리즘을 3절에서 제시하고, LOOKUP과 CLUSTER2를 결합하여 CLUSTER3를 4절에서 제시한다. 5절에서는 향후 과제를 제시한다.

3. CLUSTER를 향상시킨 CLUSTER2 알고리즘

논문 [3]의 마지막 결론 부분에 CLUSTER 알고리즘이 제시되어 있다. 이 알고리즘은 본 논문에서 중요한 역할을 하므로 이를 먼저 살펴보고, 이것의 질문 응답 시간을 향상시키는 방법 CLUSTER2를 제시한다.

3.1 CLUSTER 알고리즘 [3]

먼저, $N=n(n+1)/2$라 하자. $1\le j\le k\le n$인 모든 쌍 j,k에 대해서 $\mu(A(j,k))$를 계산하고, 이를 j,k와 함께 튜플 $(\mu(A(j,k)),j,k)$에 저장한다. 모두 N개의 튜플이

만들어지는데, 이들을 첫째 값 $\mu(A(j,k))$이 커지는 순서로 소트한다. 이렇게 소트된 튜플들을 (v_i, p'_i, q'_i)로 표시하자. 단, $1 \le i \le N$이고, $v_1 \le v_2 \le \cdots \le v_N$이다. 튜플 (v_i, p'_i, q'_i)가 의미하는 것은 인덱스 p'_i에서 시작하여 인덱스 q'_i에서 끝나는 A의 서브스트링의 질량이 v_i라는 것이다. 즉, 구간 $[p'_i, q'_i]$가 질량 v_i를 만든다.

```
i=j=0;
while( i<n ) {
   j++;
   R_j=∅;
   while( i<n ) {
      i++;
      R_j=R_j∪{p'_i};      -----(*)
      if |R_j|≥B then { r_j=v_i; break;}
   }
}
r_j=v_n;
```

그림 1 논문 [3]에서 r_j와 R_j를 구하는 알고리즘

그림 1의 알고리즘을 수행하여 소트된 리스트를 L개의 블록 $R_1, R_2, \cdots, R_L$로 나눈다. 알고리즘의 아이디어는 현재 R_j의 크기가 B 미만이면 p'_i를 R_j에 합집합 시키고, B 이상이면 바로 R_j는 닫고 다음 집합 R_{j+1}을 시작한다. 정확한 B는 나중에 정한다. R_j에 대한 유의할 점은 다음과 같다.

(i) R_j는 집합이다. 따라서, R_j의 원소들은 서로 다르다.

(ii) R_j에는 p'_i만 저장하는 것이지 i는 저장하지 않는다.

(iii) $|R_j| \le B$이다.

문장 (*)에서 인덱스 p'_i만 더해지고 있지만 개념적으로는 튜플 (v_i, p'_i, q'_i)가 R_j에 더해진 것이라 생각하면 된다. r_j는 R_j에 포함된 튜플들 중에서 가장 큰 v_i값을 나타낸다 것은 쉽게 알 수 있다. 즉, $r_1, r_2, \cdots, r_L$에 의해서 L개의 블록 $R_1, R_2, \cdots, R_L$으로 나누어진다. L은 B에 따라서 달라진다. 여기까지가 사전 처리 단계이다.

질문 응답을 하려면 먼저 $r_1, r_2, \cdots, r_L$를 이진 탐색하여 $r_{j-1} < M \le r_j$인 j를 구한다. $r_0 = 0$으로 둔다. 블록 R_j 안에 있는 각 p'_i에 대해서 그것을 시작 인덱스로 하는 A의 서브스트링 중에서 질량 M인 것이 있는지를 조사한다. 즉, $\mu(A(p'_i, p'_i))$, $\mu(A(p'_i, p'_i+1))$, ..., $\mu(A(p'_i, n))$ 중에 M이 있는지를 조사한다. 이를 위해서 스트링 A의 모든 서픽스 (suffix)들의 질량 $\mu(A(1,n))$, $\mu(A(2,n))$, ..., $\mu(A(n,n))$들을 따로 저장한다. $\mu(A(p'_i, x)) = \mu(A(p'_i, n)) - \mu(A(x+1, n))$ 이므로 이진 탐색을 이용하여 M의 존재여부를 알 수 있다. 저장하는 자료구조는 서픽스들의 질량을 저장하려면 $O(n)$ 만큼의 메모리, $r_1, r_2, \cdots, r_L$을 저장하려면 $O(L)$ 만큼의 메모리, $R_1, R_2, \cdots, R_L$을 저장하려면 $O(BL)$ 만큼의 메모리가 필요하다. 전체로는 $O(n+BL)$만큼의 메모리를 사용한다. 응답시간은 R_j를 찾는 이진탐색에 $O(\log L)$ 시간, R_j에 있는 각 p'_i에 대해서 이진탐색을 하므로 모두 $O(B \log n)$ 시간 걸린다. 합하면 $O(\log L + B \log n)$ 시간이다.

3.2 CLUSTER2 알고리즘

우리가 제시하고자 하는 알고리즘 CLUSTER2의 사전 처리 단계는 CLUSTER과 거의 같은데, 다른 점은 그림 1의 알고리즘 중에서 (*) 문장을

$$R_j = R_j \cup \{p'_i\} \cup \{q'_i\};$$

으로 바꾸는 것이다. 즉, 시작 인덱스 (p'_i) 뿐 아니라 끝 인덱스 (q'_i)도 포함시킨다. 이 경우의 유의할 점은

(i)은 앞서와 같고,

(ii) R_j에는 p'_i와 q'_i 값만 저장하는 것이지 i는 저장하지 않는다. 즉, 구간에 대한 정보는 포함되지 않는다.

(iii) $|R_j| \le B+1$이다.

또 하나 다른 점은 각 R_j에 있는 인덱스들을 크기 순으로 소트한다는 것이다. 소트 후, R_j에 있는 각 인덱스에 대해서 시작 인덱스 또는 끝 인덱스라는 표시를 한다. 인덱스가 (*) 문장에 의해서 R_j에 들어 올 때, p'_i로 온 것이면 시작 인덱스이고, q_i로 온 것이면 끝 인덱스이다. 물론, 어떤 인덱스는 시작 인덱스와 끝 인덱스 둘 다 될 수도 있다. 비록 R_j에 저장하는 데이터가 다르기 때문에 CLUSTER와 CLUSTER2가 사용하는 메모리의 정확한 값은 다르지만 asymptotic 값은 둘 다 같다는 것은 어렵지 않게 확인할 수 있다.

다음으로 주어진 질량 M인 질문에 대해서 답하는 방법을 설명하면, 먼저 $r_1, r_2, \cdots, r_L$에서 이진탐색을 수행해서 $r_{j-1} < M \le r_j$인 j를 찾는다. 역시 $r_0 = 0$이라 둔다. 이제 M을 찾으려면 블록 (집합) R_j를 조사하면 된다. R_j에는 (서로 다른) 인덱스들이 크기 순으로 소트되어 있고, 각 인덱스가 시작 인덱스인지 끝 인덱스인지 아니면, 둘 다인지 표시되어 있다. 알고리즘은 R_j에 있

는 시작 인덱스 s와 끝 인덱스 t 쌍에 대해서 ($s \le t$), $\mu(A(s,t))$가 M과 같은가를 조사한다. 모든 s, t 쌍을 다 조사하려면 $O(|R_j|^2)$ 시간이 걸리지만, 시작 인덱스와 끝 인덱스의 조사 방법을 잘 조정하면 전체적으로 $O(|R_j|)$ 쌍만 조사하면 된다. 유의할 점은 어느 인덱스가 시작과 끝 인덱스 둘 다인 경우는, 한번은 시작 인덱스로 또 한번은 끝 인덱스로서의 역할을 해야 한다는 것이다.

(a) 먼저, $s = R_j$에 있는 첫 시작 인덱스, $t = R_j$에 있는 첫 끝 인덱스로 놓는다.

(b) s와 t 둘 중 하나라도 undefined이면, "No"를 출력하고 알고리즘을 마친다. $NEXT(s)$는 R_j에서 s 바로 다음에 오는 시작 인덱스, $NEXT(t)$는 R_j에서 t 바로 다음에 오는 끝 인덱스로 놓는다. s 바로 다음에 오는 시작 인덱스가 없는 경우는 $NEXT(s)$는 undefined가 된다. $NEXT(t)$의 경우도 비슷하게 정의한다.

(c) $\mu(A(s,t)) = M$인 경우: 인덱스 (s,t) 쌍을 출력하고 알고리즘을 마친다.

(d) $\mu(A(s,t)) > M$인 경우: $s = NEXT(s)$를 하고, (b)로 간다.

(e) $\mu(A(s,t)) < M$인 경우: $t = NEXT(t)$를 하고, (b)로 간다.

위 질문 응답 방법의 정확성을 확인하는 것은 어렵지 않다. 현재 s와 t가 이루는 서브스트링의 질량 $\mu(A(s,t))$가 M과 같으면 원하는 것을 찾았으므로 알고리즘의 수행을 마치면 되고 ((c) 경우), $\mu(A(s,t)) > M$이면 s와 t가 너무 떨어져 있으므로 $s = NEXT(s)$를 해야 하고 ((d) 경우), 반대이면 $t = NEXT(t)$를 수행하면 된다 ((e) 경우). 매번 s 또는 t 둘 중 하나는 반드시 증가하므로 응답 시간은 $O(|R_j|) = O(B)$ 이다. $\mu(A(s,t))$는 3.1절에서처럼 A의 서픽스들의 질량을 저장한 배열을 이용하면 $O(1)$ 시간에 계산할 수 있다.

따라서, 알고리즘 CLUSTER2가 저장하는 자료구조는 서픽스들의 질량, $r_1, r_2, \cdots, r_L$과 $R_1, R_2, \cdots, R_L$로서 메모리양은 $O(n + BL)$이고, 질문응답 시간은 R_j를 찾는 이진탐색에 $O(\log L)$ 시간, R_j를 탐색하는데 $O(B)$ 시간 걸리므로 합쳐서 $O(\log L + B)$이다. CLUSTER와 비교하면 메모리는 동일하지만, 응답시간은 $O(\log L + B \log n)$에서 $O(\log L + B)$으로 줄였다.

4. CLUSTER2와 LOOKUP을 결합한 CLUSTER3 알고리즘

본 절에서는 논문 [3]의 LOOKUP 알고리즘의 일부 아이디어와 앞 절의 CLUSTER2를 결합하여, 응답시간은 LOOKUP과 같이 유지하면서 메모리 양만 줄이는 알고리즘 CLUSTER3를 제시한다.

4.1 사전 처리

본 절에서는 사전처리 과정을 설명한다. 먼저, A를 길이 b이면서 겹치지 않는 서브스트링들로 나눠서 이들을 순서대로 $A_1, A_2, \cdots, A_m$라 한다. 마지막 서브스트링 A_m은 길이가 b 미만일 수도 있지만 설명을 간단히 하기 위해서 A_m 역시 길이가 b라고 가정한다. 그러면 $A_i \in \Sigma^b$이고, $m = \lceil \frac{n}{b} \rceil$이다. 이 서브스트링들을 래딕스 소트 (radix sort)하여 중복은 모두 없애고 서로 다른 것들만을 모은다. 이들을 $A'_1, A'_2, \cdots, A'_l$라 하자. 당연히 $l \le m$과 $l \le |\Sigma|^b$이다. 모든 $1 \le i \le m$에 대해서 $A_i = A'_{j(i)}$인 $j(i)$를 계산하여 저장한다. 또, $\mu(A_1), \mu(A_2), \cdots, \mu(A_m)$도 저장한다. b는 n에 대한 식인데 정확한 것은 나중에 정한다.

CLUSTER2와 동일하게 N개의 튜플 $(v_i, p'_i, q'_i) = (\mu(A(j,k)), j, k)$을 구한다. 각 튜플마다 $p_i = \lceil \frac{p'_i}{b} \rceil$와 $q_i = \lceil \frac{q'_i}{b} \rceil$를 계산하여, 다른 튜플 (v_i, p_i, q_i)을 만든다. m을 앞에서처럼 $m = \lceil \frac{n}{b} \rceil$으로 두면, $p_i, q_i \in \{1, 2, \cdots, m\}$가 된다. 튜플 (v_i, p_i, q_i)가 의미하는 것은 질량 v_i인 서브스트링이 A_{p_i}의 한 원소에서 시작하여 A_{q_i}의 한 원소에서 끝난다는 것이다. p'_i와 q'_i는 구간의 시작과 끝을 정확하게 나타내지만, p_i와 q_i를 사용하면 길이 b인 서브스트링 단위로 정확도가 떨어진다. 부정확하지만 앞으로는 p_i, q_i만을 구간의 시작과 끝 인덱스로 사용한다. 이렇게 부정확한 시작, 끝 인덱스를 사용함으로써 나중에 B값을 줄일 수 있어서 알고리즘의 복잡도 향상에 기여한다.

튜플 (v_i, p_i, q_i)들을 v_i 값에 따라 크기 순으로 소트하여 $v_1 \le v_2 \le \cdots \le v_N$가 되도록 한다. 이들을 그대로 저장하려면 메모리가 $O(n^2)$가 필요하므로, CLUSTER와 CLUSTER2처럼 메모리를 줄이기 위해서 그림 2의 프로그램에 의해서 L개의 블록 $R_1, R_2, \cdots, R_L$로 나눈다. 이 알고리즘이 앞의 CLUSTER2 경우와 다른 점은 p'_i와 q'_i 대신 p_i와 q_i를 (**) 문장에서 사용한다는 것뿐이다. CLUSTER2 경우와 비슷하게, v_i 값 순서대로 소트된 튜플들이 $r_1, r_2, \cdots, r_L$에 의해서 L개의 블록 $R_1, R_2, \cdots, R_L$으로 나누어진다. R_j에 있는 인덱스들을 크기 순으로 소트하고, R_j의 각 인덱스에 대해서 시작 또는 끝 인덱스라는 표시를 한다.

```
i=j=0;
while( i<n ) {
   j++;
   R_j=∅;
   while( i<n ) {
      i++;
      R_j=R_j∪{p_i}∪{q_i};      -----(**)
      if |R_j|≥B then { r_j=v_i; break;}
   }
}
r_j=v_n;
```

그림 2 r_j와 R_j를 구하는 알고리즘

지금까지의 과정을 다시 생각해 보면, 튜플 (v_i, p_i, q_i)들을 모두 저장하는 것을 피하기 위해서, N개의 v_i들 중에서 L개만을 일정 간격으로 선택하여 저장한다. 구간 $[p_i, q_i]$들에 대한 정보는 모두 없어지고 시작 인덱스들과 끝 인덱스들만을 $R_1, R_2, \cdots, R_L$에 나눠서 저장한다. 나중에 질문 응답을 할 때 비로소 구간에 대한 정보를 복원하여 이용한다. 또한, 정확한 p'_i, q'_i 대신에 부정확한 p_i, q_i을 사용하므로, 이를 보완하기 위하여 아래 방법으로 $l \times l$ 크기의 이차원 배열 T를 만든다. l은 앞에서 언급한 $A'_1, A'_2, \cdots, A'_l$에서 온다.

T의 각 원소 $T(i,j)$는 길이 $(b+1)^2$인 배열이다. A'_i는 길이가 b이므로 공스트링을 포함하여 모두 $b+1$개의 서픽스를 가지고 있다. 이들의 질량을 커지는 순서대로 $x_0, x_1, \cdots, x_b$라 하자. A'_j는 $b+1$개의 프리픽스(prefix)를 가지고 있는데, 이들의 질량을 $y_0, y_1, \cdots, y_b$라 하자. 모든 $0 \le h, k \le b$에 대해서 $x_h + y_k$를 구한 후, 이들을 소트하여 $T(i,j)$에 저장한다. $T(i,j)$를 이용하면 어떤 값 M에 대해서 $M = x_h + y_k$가 되는 x_h와 y_k가 있는지를 $O(\log b)$ 시간에 이진탐색을 해서 알 수 있다. 즉, A'_i의 어느 서픽스와 A'_j의 어느 프리픽스를 연결하면 질량 M을 이루는가를 알아낼 수 있다.

T를 만드는 아이디어는 논문 [3]의 LOOKUP 알고리즘에서 온 것이다. 가장 큰 차이점은 $T(i,j)$인데, 우리는 $x_h + y_k$들을 소트한 것이지만, LOOKUP에서는 $x_h - y_k$들을 소트한 것이다. 물론 질문응답 방법이 다르기 때문에 $T(i,j)$에 저장되는 데이터가 다른 것은 당연하다. 또, 그 논문에서는 T의 크기가 $|\Sigma|^b \times |\Sigma|^b$이고 우리의 경우는 $l \times l$ 이다.

지금까지 만든 자료구조가 사용하는 메모리 양을 조사해 보면, T는 $O(l^2 b^2)$ 만큼, $r_1, r_2, \cdots, r_L$는 $O(L)$ 만큼, $R_1, R_2, \cdots, R_L$는 모두 합해서 $O(BL)$ 만큼, $j(1), j(2), \cdots, j(m)$와 $\mu(A_1), \mu(A_2), \cdots, \mu(A_m)$는 모두해서 $m = O(n/b)$ 만큼의 메모리를 사용하므로, 전체적으로 사용하는 메모리의 총량은 $O(l^2 b^2 + n/b + BL)$이다.

4.2 질문 응답

주어진 질량 M인 질문에 대해서 답하는 방법을 설명하면, 3절에서와 같이, 먼저 이진탐색을 수행해서 $r_{j-1} < M \le r_j$인 j를 찾아서 R_j를 조사하면 된다. 알고리즘은 R_j에 있는 시작 인덱스 s와 끝 인덱스 t 쌍에 대해서 ($s \le t$), A_s의 한 원소부터 A_t의 한 원소에 이르는 서브 스트링의 질량이 M이 되는가를 조사하는 방법으로 수행한다. 시작 인덱스와 끝 인덱스의 조사 방법을 CLUSTER2처럼 하면 $O(|R_j|)$ 쌍만 조사하면 된다. 눈여겨볼 점은 시작 (s)과 끝 (t) 인덱스의 부정확성 때문에 M이 존재하는지를 조사하려면 $T(j(s), j(t))$를 탐색해야 한다는 것과 s와 t를 전진시킬 때 조사할 조건이 더 복잡하다는 것이다. 역시, 유의할 점은 어느 인덱스가 시작과 끝 인덱스 둘 다인 경우는, 한번은 시작 인덱스로 또 한번은 끝 인덱스로서의 역할을 해야 한다.

(a) 먼저, $s = R_j$의 첫 시작 인덱스, $t = R_j$의 첫 끝 인덱스로 놓는다.

(b) s와 t 둘 중 하나라도 undefined이면, "No"를 출력하고 알고리즘을 마친다. $NEXT(s)$와 $NEXT(t)$는 CLUSTER2에서와 동일하게 정의한다.

(c) $X = \mu(A_s) + \mu(A_{s+1}) + \cdots + \mu(A_t)$을 계산한다.

또, $Y = \max(X - \mu(A_s) - \mu(A_t), 0)$을 계산한다. Y는 X에서 양쪽 항들을 뺀 것이다. $Z = \mu(A_{NEXT(s)}) + \mu(A_{NEXT(s)+1}) + \cdots + \mu(A_{NEXT(t)-1})$도 계산한다.

(d) $Y < M \le X$인 경우: (i) $M - Y$가 $T(j(s), j(t))$에 있는가를 이진탐색으로 조사하여 있으면, 해당 인덱스 쌍을 출력하고 알고리즘을 마친다. (ii) 없으면, 다음을 수행한다.

(ii-1) $NEXT(s) > t$ 이면 $t = NEXT(t)$를 하고 (b)로 간다.

(ii-2) ($NEXT(s) \le t$) & ($M \le Z$)이면 $s = NEXT(s)$를 하고 (b)로 간다.

(ii-3) ($NEXT(s) \le t$) & ($M > Z$)이면 $t = NEXT(t)$를 하고 (b)로 간다.

(e) $M \le Y$인 경우: $s = NEXT(s)$를 하고, (b)로 간다.

(f) $X < M$인 경우: $t = NEXT(t)$를 하고, (b)로 간다.

먼저 이 알고리즘이 정확하게 동작하는지를 증명하기 위해, 과연 이 알고리즘이 $Y < M \le X$을 만족하는 모든

(s, t) 쌍을 조사하는지를 살펴보자. 알고리즘이 스텝 (b)를 수행할 때마다 (s, t) 쌍을 기록한 것을 $(s_1, t_1), (s_2, t_2), \cdots$로 표시하고, (s_i, t_i)때 스텝 (c)에서 계산한 값을 X_i, Y_i, Z_i라 표시하자. 그리고 $s', t' \in R_j$인 (s', t') 쌍에 대해서 (c)의 식으로 계산한 값을 각각 X', Y', Z'라하고, $Y' < M \le X'$가 성립한다고 가정하자. 어떤 i에 대해서 $(s', t') = (s_i, t_i)$가 성립함을 보이면 된다. s'이 $s_1, s_2, \cdots$에서 처음 나오는 곳을 k라하고, t'이 $t_1, t_2, \cdots$에서 처음 나오는 곳을 l이라 한다. 즉, $s' = s_k$이며 $i < k$인 어떤 i에 대해서도 $s' \neq s_i$이고, $t' = t_l$이며 $i < l$인 어떤 i에 대해서도 $t' \neq t_i$이다. 만약, $k = l$이면 $(s', t') = (s_k, t_k)$가 성립한다.

(1) $k < l$이라 가정하자 (그림 3 참조). 그러면, $Y_k < Y'$이므로 $Y_k < M$이다. 만약 $X_k < M$이면, (f)에서 $t = NEXT(t)$가 된다. 아니면, (d)를 수행하게 되는데, (ii)일 경우 (ii-1)이 성립되면 $t = NEXT(t)$를 수행한다. (ii-1)가 성립하지 않으면, $Z_k \le Y'$이므로 $Z_k < M$이 된다. 따라서 (ii-3)에서 $t = NEXT(t)$를 수행한다. 즉, 어느 경우든 $(s_{k+1}, t_{k+1}) = (s_k, NEXT(t_k))$이다. $k+1 < l$이면, 다시 (1)를 반복해서 $(s_{k+2}, t_{k+2}) = (s_k, NEXT(t_{k+1}))$가 된다. 이런 식으로 $k+i < l$이면 $(s_{k+i}, t_{k+i}) = (s_k, NEXT(t_{k+i-1}))$가 되므로, $k+i = l$가 될 때까지 i를 계속 증가한다. 그러면, $(s_l, t_l) = (s_k, t_l) = (s', t')$가 된다.

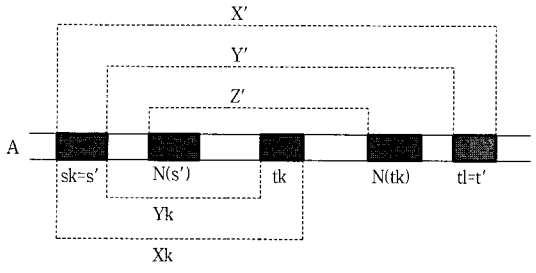

그림 3 $k < l$인 경우

(2) $k > l$이라 가정하자. (그림 4 참조) 그러면, $X_l > X'$이므로, $X_l > M$가 된다. 만약 $M \le Y_l$이면, (e)에서 $s = NEXT(s)$가 된다. 아니면, (d)를 수행하게 되는데, (ii)로 간 경우를 보자. s_k가 s_l과 t_l 사이에 있으므로, (ii-1)은 성립하지 않는다. $Z_l \ge X'$이므로 $Z_l \ge M$가 된다. 따라서 (ii-2)에서 $s = NEXT(s)$를 수행한다. 어느 경우든 $(s_{l+1}, t_{l+1}) = (NEXT(s_l), t_l)$가 된다. (1)에서와 비슷하게 $(s_{l+i}, t_{l+i}) = (NEXT(s_{l+i-1}), t_l)$가 $l+i = k$일 때까지 성립한다. 그때, $(s_k, t_k) = (s_k, t_l) = (s', t')$가 된다.

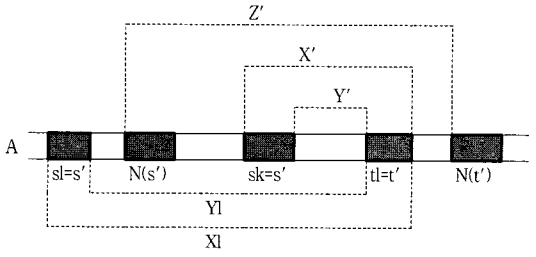

그림 4 $k > l$인 경우

즉, $(s', t') = (s_i, t_i)$가 성립하는 i가 존재한다. 따라서, R_j에 있는 $Y < M \le X$인 모든 (s, t) 쌍을 이 알고리즘이 조사하므로, R_j에 답이 있으면 꼭 찾아낸다.

질문 응답 시간을 분석하면, 먼저 $r_1, r_2, \cdots, r_L$을 이진 탐색하는 것은 $O(\log L)$ 시간이 걸린다. (b)-(f)를 한번 수행할 때마다 $s = NEXT(s)$ 또는 $t = NEXT(t)$를 꼭 수행하고, R_j는 $O(B)$개의 인덱스를 가지고 있으므로, (b)-(f)는 모두 $O(B)$번 수행된다. (d)에서 $T(j(s), j(t))$를 이진 탐색하므로 $O(\log b)$ 시간이 걸린다. 또, X_{i+1}는 X_i에 s가 전진한 만큼의 질량을 빼거나 t가 전진한 만큼의 질량을 더하면 되고, 비슷하게 Z_i도 갱신할 수 있으므로, $X_1, X_2, \cdots$와 $Z_1, Z_2, \cdots$ 모두를 $O(B)$ 시간에 계산할 수 있다. 따라서 전체 질문 응답 시간은 $O(\log L + B \log b)$이다.

4.3 성능 분석

이제, CLUSTER3 알고리즘의 성능을 알아보기 위해서 지금까지 미뤘던 b, B, L 값들을 정해 보자. 자료구조가 사용하는 메모리 양은 $O(l^2 b^2 + n/b + BL)$이고, 이를 이용한 질문응답 시간은 $O(\log L + B \log b)$이다. 둘을 n에 대한 식으로 표현하기 위해서 $b = \frac{1}{4} \log_{|\Sigma|} n$으로 놓는다. 그러면, $l^2 b^2 \le |\Sigma|^{2b} b^2 = \sqrt{n} \log^2 n$이다. 메모리 양은 $O(\sqrt{n} \log^2 n + \frac{n}{\log n} + BL)$이 돼서 $O(\frac{n}{\log n} + BL)$이 되고, 질문 응답 시간은 $O(\log L + B \log \log n)$이 된다.

여기까지의 결과를 앞의 CLUSTER2와 비교하면, 메모리 양은 $O(n + BL)$에서 $O(\frac{n}{\log n} + BL)$으로 줄었고, 질문응답 시간은 $O(\log L + B)$에서 $O(\log L + B \log \log n)$으로 늘었다.

B와 L 값을 어떻게 정하느냐에 따라 CLUSTER3의 성능이 달라진다. B를 먼저 정하면 그림 2의 알고리즘에

의해서 L은 정해진다. 만약, $B=\frac{n}{\log n}$으로 두게 되면, $L=\theta(1)$이 된다. 이 경우, CLUSTER3의 메모리 양은 $O(\frac{n}{\log n})$이고 질문응답 시간은 $O(\frac{n\log\log n}{\log n})$이다. 이것은 LOOKUP과 비교하면, 같은 질문응답 시간을 사용하지만 메모리는 $O(n)$에서 $O(\frac{n}{\log n})$으로 줄인 것이다. 즉, 다음 정리를 증명한 것이다.

정리 1: 메모리 양은 $O(\frac{n}{\log n})$이고 질문응답 시간은 $O(\frac{n\log\log n}{\log n})$인 탐색 자료구조를 만들 수 있다.

5. 결론 및 향후 과제

본 논문에서는 기존의 CLUSTER 알고리즘을 향상시키기 위해서 CLUSTER2와 CLUSTER3를 차례로 개발하였다. 이들의 복잡도를 비교해 보면 표 1과 같다. 특히 CLUSTER3의 경우 $B=\frac{n}{\log n}$로 둠으로써 주 결과인 정리 1을 얻을 수 있었다. 언급할 것은 CLUSTER와 CLUSTER2에서는 B가 가질 수 있는 최대가 n이지만 CLUSTER3의 경우는 $n/\log n$이다.

표 1 세 알고리즘의 복잡도 비교

	메모리 양	질문응답 시간
CLUSTER	$O(n+BL)$	$O(\log L+B\log n)$
CLUSTER2	$O(n+BL)$	$O(\log L+B)$
CLUSTER3	$O(\frac{n}{\log n}+BL)$	$O(\log L+B\log\log n)$

향후 연구과제로서, 세 알고리즘의 정확한 성능은 모두 B와 L 사이의 관계를 정확하게 파악함으로써 밝힐 수 있다. 그러나, B에 따라 L이 어떻게 변하는가를 이론적으로 규명하는 것은 [3]에서 언급한 것처럼 쉽지 않을 것 같다. 또, 이를 프로그램 실험으로 B값에 따른 L값을 구해 봤으나 이 역시 B와 L 사이의 관계를 유추하는데 별로 도움이 되지 않았다.

참 고 문 헌

[1] Protein Data Bank, http://www.rcsb.org/pdb/.
[2] SWISS-PROT, http://www.expasy.ch/sprot/sprot-top.html.
[3] M. Cieliebak, T. Erlebach, Z. Liptak, J. Stoye, and E. Welzl, Algorithmic complexity of protein identification: Combinatorics of weighted strings, Technical Report No. 361, Computer Science Department, ETH Zurich, August 2001.
[4] D. Gusfield, *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1997.

김 성 권
정보과학회논문지 : 시스템 및 이론
제 29 권 제 3 호 참조