공간 분할 방법을 이용한 최적 서열정렬 알고리즘

(Optimal Sequence Alignment Algorithm Using Space Division Technique)

안 희 국 † 노 희 영 ††
(Heui Kook Ahn) (Hi Young Roh)

요 약 두 서열 A와 B간의 최적정렬을 찾는 문제는 동적프로그래밍 알고리즘을 사용하여 효과적으로 해결 될 수 있다. 하지만, 길이가 각각 m, n인 두 서열, S_1, S_2를 정렬하기 위해서는 $O(m*n)$의 시간과 공간 복잡도를 갖기 때문에 서열의 길이가 길어질 경우에는 시간과 공간 비용 문제로 인해 적용 할 수 없게 된다. 실제 계산상에 제한요소로 작용하는 공간비용 문제를 해결하기 위해 Hirschberg에 의해 제시된 선형공간 알고리즘은 이 문제를 $O(n*m)$의 시간복잡도와 $O(n+m)$의 공간복잡도로서 해결하였다. 컴퓨터 기술의 발전으로 CPU의 처리속도가 향상되고, 사용가능한 주기억장치의 공간이 확대됨에 따라, 기억공간은 더 사용하더라도 처리속도는 높일 수 있는 방법이 필요하다. 이를 위해, 본 논문에서는 공간 분할 방법을 통하여 공간 소모는 선형공간 알고리즘보다 많지만, 처리 속도는 빠른 $O(n*m)$의 시간과 $O(n+m)$의 공간 비용을 갖는 알고리즘을 제안한다. 또한 분할 시 서열의 길이변화에 따른 분할 수(d) 문제를 일반화하고, 입/출구 노드 개념을 이용하여 불필요한 연산을 제거하였다. 선형공간 알고리즘이 (m+n)의 공간으로 $2*m*n$에 가까운 속도를 갖는데 비해, 본 알고리즘은 $(m+n)*d$의 공간으로 $m*n$에 가까운 결과를 보임을 증명과 실험결과로부터 확인한다.

키워드 : 서열 최적정렬, 동적프로그래밍 알고리즘, 선형공간 알고리즘, 분할일반화

Abstract The problem of finding an optimal alignment between sequence A and B can be solved by dynamic programming algorithm(DPA) efficiently. But, if the length of string was longer, the problem might not be solvable because it requires $O(m*n)$ time and space complexity.(where, m=|A|, n=|B|) For space, Hirschberg developed a linear space and quadratic time algorithm, so computer memory was no longer a limiting factor for long sequences. As computers's processor and memory become faster and larger, a method is needed to speed processing up, although which uses more space. For this purpose, we present an algorithm which will solve the problem in quadratic time and linear space. By using division method, It computes optimal alignment faster than LSA, although requires more memory. We generalized the algorithm about division problem for not being divided into integer and pruned additional space by entry/exit node concept. Through the proofness and experiment, we identified that our algorithm uses $d*(m+n)$ space and a little more $(m*n)$ time faster than LSA.

Key words : Sequence Optimal Alignment, Dynamic Programming Algorithm, Linear Space Algorithm, Division Generalization

1. 서 론

서열정렬이라 함은 두 개의 서열을 순서를 유지하고, 공백(space)을 허용하면서 나열하는 것을 말하며, 두 서열간의 각 심벌들을 점수함수(일치, 불일치, 갭일 때 값을 부여하기위한 함수)를 기준으로 1:1로 점수화 시켰을 때, 점수값의 합이 최대가 되는 서열을 최적정렬이라 한다[1,2].

생물서열(DNA, amino acid)에서 최적정렬의 중요성은 공통조상으로부터 분화된 상동유전자(homologous gene)는 구조적 기능적면에서 서로 유사성을 갖는다는 데 있다. 즉, 새로운 유전자 A와 이미 알려진 유전자 B

† 정 회 원 : 강원대학교 컴퓨터과학과
basic@kangwon.ac.kr
†† 정 회 원 : 강원대학교 컴퓨터과학과 교수
rohhy@kangwon.ac.kr
논문접수 : 2006년 10월 9일
심사완료 : 2007년 2월 22일

가 유사성을 갖는다면, 두 유전자는 서로 구조와 기능면에서 비슷하다는 결론을 추론할 수가 있으며, 이러한 유사성을 판단하기 위한 방법으로 사용되는 것이 최적정렬이다[3,4].

2006년 8월 현재 NCBI(National Center for Biotechnology Institute)의 유전자은행에 저장된 서열정보는 61,132,599개의 서열에 65,369,091,950개의 염기가 저장되어 있고, 그 수는 계속적으로 증가되고 있는 상황이다[5]. 또한 최근에는 서열의 길이가 수십억 개까지 될 수 있는 두 개의 전체 게놈을 비교하는 것이 연구자들 사이의 초점이 되고 있다[6-8]. 즉, 큰 서열로부터 최적정렬을 찾는 요구가 증대됨에 따라 컴퓨터를 이용한 최적정렬 계산 시, 요구되는 시간 및 공간 비용이 증가되고 있으며, 이를 효율적으로 감소시킬 방법 및 필요성이 증대되고 있는 상황이다[9,10].

최적정렬을 구하는 문제는 Needleman & Wunsch[70]의 동적 프로그래밍 알고리즘(Dynamic Programming Algorithm :DPA.)을 통해 효과적으로 구해질 수 있다[11]. 하지만, 2차원의 전체 매트릭스를 구성해야하기 때문에 $O(m*n)$의 시간과 공간 복잡도를 갖게 된다(m=|S1|, n=|S2|). 예를 들어, 길이 10,000의 서열 S1, S2는(계수: 1μs/operation, 1Byte/symbol) 100sec 시간과 100M 공간이 필요하다. 즉, 서열의 크기가 커질 경우, 시간보다는 공간 비용이 제한요소로서 작용하게 되어 문제를 풀 수 없게 된다.

정방형의 공간 비용을 선형공간으로 줄이기 위해 Hirschberg[75]는 $O(m+n)$의 선형공간 알고리즘(Linear Space Algorithm :LSA.)을 제안하였다. 비록, 시간비용면에서는 약 2배정도의 시간을 필요로 하지만, 여전히 $O(m*n)$의 범위 안에 포함된다[12]. 예를 들어, DPA의 경우처럼, 길이 10,000의 서열 S1, S2는(계수: 1μs/operation, 1Byte/symbol) 200sec 시간을 소요하지만, 10K 공간만을 사용한다. 즉, 서열간의 최적정렬을 구하는 문제에 있어서 제한요소로 작용했던, 공간비용 문제는 LSA를 통해 해결되었다. 그로인해 현재 더욱 긴 서열들이 일반 워크스테이션 상에서 정렬될 수 있다. 예) 1 hour/100,000 letters. 후에 Myers와 Miller는 q>=0인 경우를 다루기 위해 LSA를 일반화하였다[13].

LSA의 장점은 $(m+n)$의 공간비용만을 사용한다는 것을 들 수 있지만, 이론적으로 항상 $2*m*n$에 가까운 시간비용을 소요한다는 것이다. 오늘날 컴퓨터의 속도향상과 함께, 기억장소에 대한 가용성도 증가하여 추가적인 공간비용을 지불하더라도, 시간적 이득을 얻을 수 있다면, 훨씬 실용적인 알고리즘이 될 수 있다. 이에, 본 논문에서는 공간분할 기법을 이용하여, 분할면의 정보만을 테이블로 유지하면서, 최적정렬을 얻을 수 있는 $O(m*n)$ 시간 복잡도와 $O(m+n)$ 공간 복잡도를 갖는 알고리즘을 제안한다. 예를 들어, 길이 10,000의 서열 S1, S2는(계수: 1μs/operation, 1Byte/symbol, d=100: 분할수) 제안된 방법을 통하여 2.01M의 공간만을 사용하고, 최악의 경우를 고려할 때, 102초 정도 만을 소요하며, 최적정렬을 얻을 수 있다.

본 논문의 아이디어는 매트릭스를 분할함으로서 생성되는 블록의 최적정렬은 전체 엔트리 값을 얻기 위해 블록의 left-top corner, left line, top line만을 필요로 한다는 데에 있다. 필요한 데이타들을 추출하고 저장하기위해 LSA처럼, 두 라인만을 유지하면서, 분할 면의 정보를 얻는다. 그리고, 이로부터 구하고자하는 블록의 엔트리 값을 얻기 위한 정보를 얻어 DPA를 시행하고, 부분 최적정렬을 얻는다. 부분 블록의 선택 및 연결은 입/출구 노드 개념을 사용함으로서 정확하고 간단하게, 부분 최적정렬들로부터 전체정렬을 얻게 된다. 본 연구에서는 추가적으로 서열이 정확히 분할수로 분할되지 않는 m%d!=0인 경우에 대하여 분할일반화를 통하여 서열 길이 및 분할수와 상관없이 제안한 알고리즘이 적용될 수 있도록 알고리즘을 설계하였다.

본 논문에서는 증명 및 실험을 통해, 제안한 최적 서열정렬 방법이 공간적 측면에서 DPA보다 상당한 공간을 줄일 수 있고, 비록 LSA보다 공간은 더 필요로 하지만, 처리시간 면에서는 DPA와 가깝게 빠르다는 것을 확인하였다.

2. 문제 정의

서열정렬은 스트링매칭의 특수한 경우로서, 생물서열의 관점에서는 진화적 변화를 표현하는 두 스트링간의 편집거리를 결정하는 과정이라 볼 수 있다[14]. 편집거리는 편집연산(삽입, 삭제, 치환)에 의해 결정되며, 각각에 대한 점수화를 통해 결정된다.

길이가 m과 n인 두 서열을 $S=s_1s_2...s_m$, $T=t_1t_2...t_n$로 하고, S와 T의 원소 쌍을 (s_i, t_j)로 놓자. 단, $i=1...m$, $j=1...n$.

[정의 1] S와 T의 원소간의 관계, (s_i, t_j), $(-, t_j)$, $(s_i, -)$를 정의하는 함수를 점수함수라 하며, σ로 표기한다. (단, '-'는 공백)

예) $\sigma(s_i, t_j)$=+2(match), -1(mismatch)
$\sigma(-, t_j)$=-1(gap),
$\sigma(s_i, -)$=-1(gap)

[정의 2] 두 서열 S와 T를 구성하는 원소들의 순서는 유지하되, 공백을 포함하면서 쌍을 구성하였을 때, 순서쌍의 집합을 정렬 $\bar{A}$라 하며, 모든 쌍에 대한 σ합을 정렬 값(A)이라 한다.

예) S=abcdab, T=abdb.→S′=abcdab, T′=ab-d-b

$\bar{A}$={(a,a), (b,b),(c,-),(d,d),(a,-),(b,b)}.

$$A=\sum_{i=1}^{l}\sigma(S'[i], T'[i]),\ 단, l=|S'|=|T'| \quad (1)$$

두 서열간의 유사도는 정렬 값, 식 (1)으로 표현되며, 생물서열의 경우 두 서열간의 구조 및 기능, 상동성 등을 추측하기위해 사용하게 된다.

[정의 3] 서열 S와 T의 최적정렬($\bar{A}$)은 두 서열에 대한 모든 가능한 정렬 중 최대정렬 값(A)을 갖는 정렬을 말한다.

위 예에서 최적정렬 $\bar{A}$는 S'=abceab, T'=ab-d-b가 되며, 최적정렬 값 A는 6이다.

두 서열로부터 최적정렬을 찾는 문제는 모든 가능한 정렬에 대해 정렬 값을 구하는 것이다. 이는 단순히, 다음 알고리즘으로 찾을 수 있다.

[ALG 1] 최적정렬을 위한 단순한 알고리즘

for all i, $0\le i\le n$, **do**
 for all subsequence A of S with |A|=i **do**
  for all subsequence B of T with |B|=i **do**
   Form an alignment that matches A[k] with B[k], $1\le k\le i$, & matches all other characters with space.
   Determine the value of this alignment;
   Retain the alignment with maximum value;
  end;
 end;
end;

[ALG 1]의 시간분석을 보면, 길이 n의 서열은 길이가 i인 부서열 $\binom{n}{i}$개를 가지므로, $\binom{n}{i}^2$개의 (A, B)쌍을 갖는다. S에는 n의 문자가 있고, 그 중, i개만이 T의 문자들과 일치하고, $n-i$개의 문자들은 일치하지 않는다. 따라서, 정렬은 길이 $n+(n-i)=2n-i$를 갖는다. 정렬시, 각 쌍의 점수를 더하고, 참조해야 하므로, 기본 연산의 총 합은 최소한 다음과 같다[15].

$$\sum_{i=0}^{n}\binom{n}{i}^2(2n-i)\ge n\sum_{i=0}^{n}\binom{n}{i}^2=n\binom{2n}{n}>2^{2n},\ \text{for } n>3 \quad (2)$$

예를 들어, n=20일 때, [ALG 1]은 최소 $2^{2n}=2^{40}$이상의 연산을 수행한다. 즉, 시간비용으로 인해 최적정렬 문제는 이 알고리즘으로 접근할 수가 없게 된다.

이러한 문제는 DPA를 통해 $O(m*n)$ 시간과 공간 비용으로 풀 수 있으며, 실제 제한요소로 작용하는 공간 복잡도는 LSA를 통해 $O(m*n)$시간과 $O(m+n)$공간비용으로 해결될 수 있다. 좀 더 정확히, $2*m*n$에 가까운 시간과 $2*m$의 공간으로 해결된다. 이를 관련연구에서 알아본다.

본 논문에서 제안하는 알고리즘은 LSA처럼, $O(m*n)$ 시간과 $O(m+n)$공간 복잡도를 갖지만, 정확히는 $d*(m+n)$의 공간과 $m*n$에 가까운 시간을 갖는다. 즉, LSA보다 공간은 더 소모하지만, 시간적인 측면에서는 훨씬 빠르게 최적정렬을 얻을 수가 있다. 적용된 방법은 분할기법을 사용하며 이와 관련된 내용은 본론에서 기술한다.

3. 관련연구

3.1 Dynamic Programming Algorithm(DPA)

1970년 Needleman & Wunsch에 의해 도입된 DPA는 문제를 분할하여 작은 문제를 해결해 나감으로서 전체문제를 해결하는 알고리즘으로서 최소의 비용을 갖는 경로를 찾거나, 작업스케줄링, 스트링 매칭 등에 이용될 수 있는 최적화 알고리즘이다. 특히, 서열정렬에 있어서 유용하게 이용될 수 있다[11,16].

서열 S와 T의 최적정렬 값은 $A(m,n)$이며, $0\le i\le m$, $0\le j\le n$ 범위내의 모든 $A(i,j)$(단, 순서유지)가 계산되어져야 한다. 이를 위해 DPA는 작은 i와 j를 갖는 이미 계산된 $A(i,j)$에 다음의 계산을 반복함으로서 모든 $A(i,j)$를 계산한다.

반복과정 :

$$A(i,j)=\max(\ A(i-1,j-1)+\sigma(S[i],T[j]),\ A(i-1,j)+\sigma(S[i],-),\ A(i,j-1)+\sigma(-,T[j])\ )$$

S의 i번째 문자가 T의 0번째 문자(공백)와 정렬되는 경우 $A(i,0)$를 고려하기위해 반복과정에 앞서 다음의 초기화과정을 시행한다.

초기화과정 :

$$A(0,0)=0$$
$$A(i,0)=A(i-1,0)+\sigma(S[i],-),\ \text{for } i>0$$
$$A(0,j)=A(0,j-1)+\sigma(-,T[j]),\ \text{for } j>0$$

즉, 초기화 과정과 반복과정을 통해 각 정렬 값 $A(i,j)$는 다음 중 하나를 보장해야한다.

1. (S[i], T[j])는 자신을 제외하고 남은 정렬 S[1]…S[i-1]과 T[1]…T[j-1]는 최적정렬 이어야 하며, 정렬 값 $A(i-1,j-1)$를 갖는다.
2. (S[i], -)는 자신을 제외하고 남은 정렬 S[1]…S[i-1]과 T[1]…T[j]는 최적정렬 이어야 하며, 정렬 값 $A(i-1,j)$를 갖는다.
3. (-, T[j])는 자신을 제외하고 남은 정렬 S[1]…S[i]과 T[1]…T[j-1]는 최적정렬 이어야 하며, 정렬 값 $A(i,j-1)$를 갖는다.

전체 알고리즘은 입력스트링으로 A_{1m}과 B_{1n}를, 출력으로 매트릭스 $\bar{A}$를 갖는 DPA함수로 표현 될 수 있다. (단, 각 엔트리 $A(i,j)$는 S_{1i},T_{1j}까지의 최적정렬 값에 해당한다.)

[ALG 2] 동적프로그래밍 알고리즘

(1) Initialization: $A(i,0) \leftarrow i*\sigma(i,-)$ $[i=0\cdots m]$;
$A(0,j) \leftarrow j*\sigma(-,j)$ $[j=0\cdots n]$;

(2) **for** $i \leftarrow 1$ **to** m **do**
begin

(3) **for** $j \leftarrow 1$ **to** n **do**
Diagonal $\leftarrow A(i-1,j-1) + \sigma(S[i],T[j])$,
Vertical $\leftarrow A(i-1,j) + \sigma(S[i],-)$,
Horizontal $\leftarrow A(i,j-1) + \sigma(-,T[j])$
$A(i,j) \leftarrow$max(Diagonal, Vertical., Horizontal)
end

(4) **write** $A(m,n)$

(5) **recovering** (m, n, $A(m,n)$, C)

[ALG 2]의 (5)는 (4)를 통해 얻은 전체 매트릭스 $A(m,n)$와 두서열 m, n으로부터 역추적과정을 통해 최적정렬 C을 얻는다. 과정은 엔트리 (m,n)의 경우, 그 이전 엔트리(m-1, n-1), (m-1, n), (m, n-1)들 중에서 $A(m,n)$에 영향을 준 엔트리를 찾아감으로서 얻어진다. (단, i=0, j=0까지)

[ALG 2]의 시간과 공간비용은 (3)이 정확히 $m*n$번 시행되며, 입력과 출력 배열은 $(m+n) + (m+1)*(n+1)$공간을 필요로 하므로, $O(m*n)$ 시간과 $O(m*n)$ 공간을 필요로 한다.

3.2 Hirschberg's Linear Space Algorithm(LSA)

[ALG 2]는 서열이 길어질 경우에 공간비용이 제한요소로 작용하므로, 공간 복잡도를 줄이기 위해 75년 Hirschberg는 O(m+n)의 공간으로 최적정렬을 구할 수 있는 알고리즘을 제안하였다[12].

LSA는 전체 문제를 가장 작은 문제를 해결할 때까지 2개로 분할해 나가면서 재귀적으로 문제를 해결하는 알고리즘이다. 방법은 DPA의 매트릭스 $\bar{A}(A(i,1), A(i,2), \ldots, A(i, n))$의 i행을 구하기 위해서는 i-1행만이 필요하다는 점과 역으로 DPA를 시행하더라도 동일한 정렬 값을 얻는다는 것에 착안하여, 2개의 행만 유지하면서, 문제를 해결해 나간다.

[ALG 3] 선형공간 순방향 알고리즘

(1) Initialization: $A(1,j) \leftarrow j*\sigma(-,j)$ $[j=0,\cdots,n]$;

(2) **for** $i \leftarrow 1$ **to** m **do**
begin

(3) $A(0,j) \leftarrow A(1,j)$ $[j=0,\cdots,n]$;

(4) **for** $j \leftarrow 1$ **to** n **do**
Diagonal $\leftarrow A(0,j-1) + \sigma(S[i],T[j])$,
Vertical $\leftarrow A(0,j) + \sigma(S[i],-)$,
Horizontal $\leftarrow A(1,j-1) + \sigma(-,T[j])$
$A(1,j) \leftarrow$max(Diagonal, Vertical., Horizontal)
end

(5) $AA(j) \leftarrow A(1,j)$ $[j=0,\cdots,n]$

선형공간 순방향 알고리즘의 결과는 최종적으로 (5)의 출력벡터 AA(j)로서 A(1,j)의 엔트리 집합이며, 최적정렬 값은 $A(1,n)$이 된다. [ALG 3]은 (4)에서 for문이 정확하게 $m*n$번 실행된다. 입출력 배열은 $m+n+$ $(n+1)$을 필요로 하고, 매트릭스는 $2(n+1)$을 필요로 한다. 따라서, $O(m*n)$시간과 $O(m+n)$공간 복잡도를 갖는다.

[ALG 3]은 [ALG 4]와 같이 역으로도 구해질 수 있다.

[ALG 4] 선형공간 역방향 알고리즘

(1) Initialization: $A(0,j) \leftarrow$(n-j)$*\sigma(-,j)$ $[j=0,\cdots,n]$;

(2) **for** $i \leftarrow m$-1 **down to** 0 **do**
begin

(3) $A(1,j) \leftarrow A(0,j)$ $[j=n,\ldots,0]$;

(4) **for** $j \leftarrow n$-1 **down to** 0 **do**
Diagonal $\leftarrow A(1,j+1) + \sigma(S[i],T[j])$,
Vertical $\leftarrow A(1,j) + \sigma(S[i],-)$,
Horizontal $\leftarrow A(0,j-1) + \sigma(-,T[j])$
$A(0,j) \leftarrow$max(Diagonal, Vertical., Horizontal)
end

(5) $AA(j) \leftarrow A(0,j)$ $[j=n,\cdots,0]$

[ALG 4]의 최종결과는 (5)의 출력벡터 AA(j)로서 A(0,j)의 엔트리 집합이며, 최적정렬 값은 A(0,0)이 된다.

[ALG 3]과 [ALG 4]를 이용하여 LSA는 전체 서열의 최적정렬(Å)를 구할 수가 있다.

[ALG 5] 최적정렬을 위한 선형공간 알고리즘

(1) If problem is trivial, solve it:
if n=0
then C$\leftarrow e$ (e: empty string)
else if m=1
then if $\exists$j$\leq$n such that A(1)=B(j)
then C$\leftarrow$A(1)
else C$\leftarrow e$

(2) Otherwise, split problem:
else begin i$\leftarrow \lfloor$ m/2 $\rfloor$;

(3) Evaluate L(i, j) and L*(i, j) [j=0$\cdots$n]:
ALG 3(i, n, A_{1i}, B_{1n}, L1);
ALG 4(m-i, n, $A_{m,i+1}$, B_{n1}, L2);

(4) Find j such that L(i, j)+L*(i, j)=L(m, n):
M$\leftarrow$max{L1(j)+L2(n-j)} ;(where, 0$\leq$j$\leq$n)
k$\leftarrow$min j such that L1(j)+L2(n-j)=M;

(5) Solve simpler problems:
ALG 5(i, k, A_{1i}, B_{1k}, C1);
ALG 5(m-i, n-k, $A_{i+1.m}$, $B_{k+1.n}$, C2);

(6) Give output:
C$\leftarrow$C1||C2;
end

[ALG 5]는 길이 m과 n의 서열 A, B를 입력받아 최적정렬을 내놓는다. ALG 3, 4, 5에서 사용된 인수들 중에서 첫 번째와 두 번째는 두서열의 각각의 길이를, 그리고, 세 번째와 네 번째는 해당길이에 해당하는 위치의 부분서열을 의미하고, 다섯 번째 인수는 출력값을 의미한다. 예를 들어, ALG 3은 입력스트링으로 A_{1m}, B_{1n}을 받고, 출력으로 벡터 L1을 내놓는다. 최종적으로 최적정렬 C는 (5)의 자신의 함수를 재귀적으로 호출함으로서 얻어지는 부분서열들을 연결함으로서 구해진다[12]. T를 원래 길이로 놓으면, 모든 재귀단계에서 하위 문제들의 총 크기는 최대, T+1/2T+1/4T…=2T이다. 즉, $O(m+n)$ 공간 복잡도를 갖는다. 계산 시간은 직접적으로 문제의 크기에 비례하므로, 단순히 최적점수를 계산하는데 필요한 시간의 2배정도를 소요한다. 즉, $O(m*n)$의 시간 복잡도를 갖는다.

4. 공간분할 방법을 이용한 알고리즘

본 논문에서는 최적정렬을 찾는 문제에 있어서, LSA가 $2*m*n$의 시간에 가깝게 수행하는데 비하여, 비록, 공간은 더 사용하더라도 $m*n$에 가까운 시간으로 해를 찾아내기 위한 알고리즘을 제안한다.

4.1 알고리즘의 설계

[정의 4] 서열 $S_{i(i=1\cdots m)}$와 $T_{j(i=1\cdots n)}$에 대해 DPA의 과정에서 생성된 매트릭스 A의 특정 엔트리를 A(i, j)라 할 때, A(i-1, j-1), A(i, j-1), A(i-1, j)를 각각 분면 q_1, q_2, q_3라 하며, A(i, j)를 분면 q_4로 정의한다.

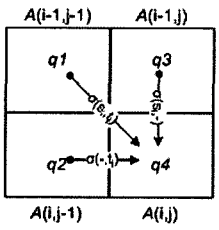

그림 1 DPA의 q_1, q_2, q_3, q_4 정의

[ALG 2]로부터 $A(i, j)$는 $A(i-1, j-1)$, $A(i, j-1)$, $A(i-1, j)$로부터 계산된 최대값을 가지므로, q_4는 q_1, q_2, q_3에 의해서만 결정된다.

[정의 5] DPA 매트릭스 A를 행과 열에 대해 정수 d로 분할함으로서 생성된 부분 매트릭스(block, B)의 행과 열 크기를 β_R, β_C라하며, $B=\{B(k,l)|k\le d,\ l\le d\}$를 블록 매트릭스로 정의한다(단, k, l은 I, j로부터 d에 의해 축소된 인덱스).

[정의 6] 블록 $B(k,l)$의 상, 하, 좌, 우 면을 각각 상선, 하선, 좌선, 우선이라 하며, 이전 블록 $B(k-1, l-1)$, $B(k, l-1)$, $B(k-1, l)$과 자신 $B(k, l)$를 각각 분면 Q_1, Q_2, Q_3, Q_4로 정의한다.

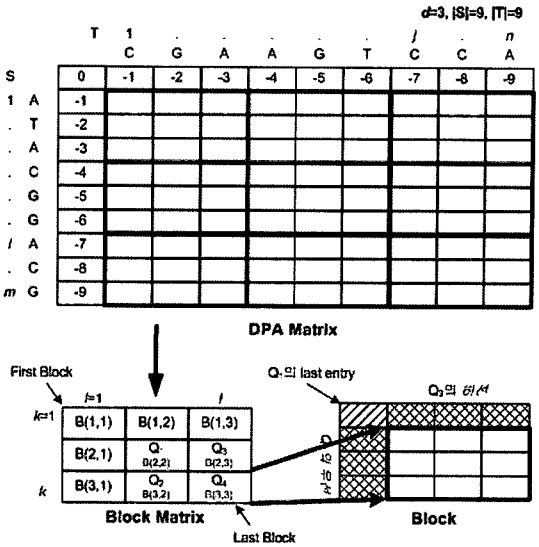

그림 2 Block matrix의 Q_1, Q_2, Q_3, Q_4 정의

[정리 1] 블록 매트릭스 B의 특정 엔트리 Q_4는 이전 엔트리 Q_1의 우선과 좌선의 마지막엔트리와 Q_2의 좌선, Q_3의 하선에 의해서만 결정된다.

증명. 블록 매트릭스 B의 구하고자하는 특정엔트리 집합은 $Q_4 = \{A(i,j)|\ i=i+\kappa_i,\ j=j+\kappa_j,$ 단, $0\le\kappa_i,\kappa_j<\beta_R,\beta_C\}$이다. Q_4를 계산하기 위해 필요한 엔트리 집합은 $R=\{A(i,j)|\ i=i+\kappa_i-1,\ j=j+\kappa_j-1,$ 단, $0\le\kappa_i,\kappa_j<\beta_R,\beta_C\}$이다. [알고리즘 2]에 의하여 엔트리집합 $R'=\{A(i,j)|i=i+\kappa_i,\ j=j+\kappa_j,\ 0<\kappa_i,\kappa_j<\beta_R,\beta_C\}\subset Q_4$이며, $R'\subset R$이므로 블록 내에서 이전 엔트리 값들로부터 계산될 수 있다. 하지만, $R-R'=\{A(i,j)|i=i+\kappa_i,\ j=j+\kappa_j,$(단, κ_i 또는 $\kappa_j=0$)}인 경우에는 Q_4를 계산하기 위해 다음 엔트리를 요구한다. 즉, $\kappa_i=0$, $\kappa_j=0$일 때, Q_1의 마지막엔트리를, $\kappa_i=0$, $\kappa_j\neq0$일 때, Q_3의 하선을, $\kappa_i\neq0$, $\kappa_j=0$일 때, Q_2의 우선을 요구한다. 따라서, 특정 블럭 B의 모든 엔트리를 구하기위해서는 Q_1의 마지막엔트리, Q_2의 우선, Q_3의 하선만 요구한다. □

본 논문에서는 DPA가 최적정렬을 구하기 위해 전체 매트릭스를 유지해야하는 문제를 [정리 1]을 이용하여 해결한다. 즉, 전체 문제를 분할하여 작은 문제에 대해 최적정렬을 구하고, 해결된 문제들을 연결함으로서 전체 문제를 해결한다. 작은 문제들을 해결하기 위하여 먼저, 요구되는 최소 엔트리들만을 분할 수 d로 나눠진 행과 열 테이블에 저장하고, 문제를 해결할 때, 이를 참조한다. 이때, DPA는 고정된 크기의 블록을 이용하여 마지막 블록부터 시작 블록까지 연결되는 모든 블록들에 대

해서 순차적으로 시행된다.

서로 다른 길이의 서열을 정수 d로 분할할 때, 정수로 나눠지지 않을 경우, 고정된 크기의 블록 안에서 문제를 해결할 수 없는 경우가 발생한다. 이를 위해 본 논문에서는 다음의 일반화정리를 적용한다.

[정리 2] 서열 S의 길이를 m, 분할수를 정수 d, 블록 테이블의 크기를 β_R라 할 때, $m/d \le m\%d$이면, 블록 크기 $\beta_R=(m/d+1)$이고, 그렇지 않으면, $\beta_R=m/d$이다. (단, /=몫, %=나머지)

증명. m=N(단, N≥0정수), $d \le m$(단, d>0정수)로 놓자. 블록크기 $\beta_R=m/d$라 하면, 마지막 블록 크기는 $\beta_{last}=m\%d$이고, $m/d \le m\%d$인 경우, $B_R<B_{last}$가 된다. 따라서, 블록 B_R안에서 B_{last}는 해결될 수 없으므로, $\beta_R=m/d+1$로 놓으면, 항상 $m/d>m\%d$가 되어 블록크기 β_R안에서 모든 블록의 최적정렬을 구할 수 있다. □

유지해야하는 분할테이블 D는 행 테이블(R_table)과 열 테이블(C_table)로 구분하여 저장된다. 이때, 행 테이블의 열 크기(D_col_i)와 열 테이블의 행 크기(D_row_j)는 다음 식으로 구해진다.

(1) D_col_i=Length_one/Block_size_one+1

(2) D_row_j=Length_two/Block_size_two+1

[식 3] 행, 열 테이블의 인덱스크기.

공간비용 축소를 위해 분할테이블을 작성하는 과정은 [ALG 3]의 H_Table을 통하여 전체 최적정렬 값을 구하는 과정에서 필요한 엔트리들만을 선택함으로서 진행된다.

[ALG 6] 행 테이블 작성을 위한 엔트리선택

begin

R_Table[i][col]←H_Table[1][j], [j=j+β_C, 0≤j≤n]

col++;

end

[ALG 7] 열 테이블 작성을 위한 엔트리선택

begin

if i%β_R=0

row++;

C_Table[row][j] ←H_Table[1][j], [j=0,...,n];

end

문제를 |S|=36, T=|17|, d=5인 경우를 예로 살펴보면 다음과 같다.

[정리 2]로부터 DPA를 통해 구해야 할 블록의 크기는 7*3이며, [정리 1]로부터 Q_1의 마지막엔트리와 Q_2의 우선, Q_3의 하선을 필요로 하므로, 그림 4의 8*4블록을 갖는다.

[ALG 3]으로부터 생성된 중간테이블은 그림 5와 같다.

[ALG 7]로부터 생성되는 열 테이블은 그림 6과 같다.

제안하는 알고리즘의 분할테이블 및 블록테이블을 생성하는 전체과정을 기술하면 다음과 같다.

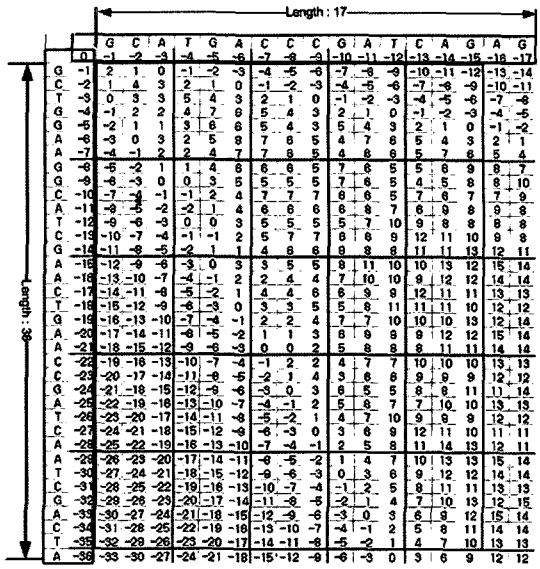

그림 3 DPA 분할 예

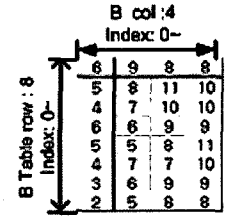

그림 4 블록테이블(B_Table)

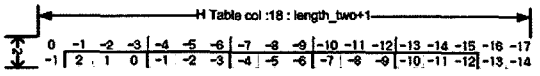

그림 5 중간 테이블(H Table)

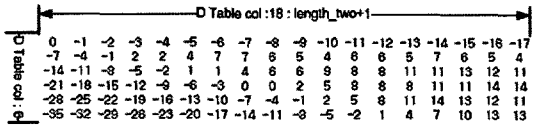

그림 6 열 테이블(C_table)

[ALG 8] 분할 및 블록테이블 생성

(1) Calculate : β_R, β_C.

(2) Create table :

R_table, C_table, B_table, H_table

(3) Initialization: $H\_Table(1,j)$ ←j*σ(−,i)[j=0,⋯,n];

(4) **for** i←1 **to** m **do**

begin

(5) $H\_Table(0,j)$←$H\_Table(1,j)$ [j=0,...,n];

(6) **for** $j \leftarrow 1$ **to** n **do**

Diagonal $\leftarrow H\_Table(0,j-1) + \sigma(S[i],T[j])$,

Vertical $\leftarrow H\_Table(0,j) + \sigma(S[i],-)$,

Horizontal $\leftarrow H\_Table(1,j-1) + \sigma(-,T[j])$

$A(1,j) \leftarrow$ max(Diagonal, Vertical, Horizontal)

(7) **ALG 6, ALG 7**

end

(8) Give output: D', B'

[ALG 8]은 길이가 m과 n인 서열, S와 T, 그리고, 분할수 d를 입력으로 받아, 분할(D') 및 블록 테이블(B')을 생성하는 과정을 나타낸 것이다.

생성된 블록과 분할테이블로부터 역추적과정을 통해 전체 최적정렬을 구하기 위해 다음을 정의한다.

[정의 7] 분할로 생성된 블록테이블(B_Table)내에서 역추적과정이 시작되는 노드를 입구노드(In-node)라 하며, 종결되는 노드(B_i=0 or B_j=0)를 출구노드(Out-node)라 한다.

[정리 3] 특정 블록(B_{p2})의 입구노드는 연결되는 이전 블록의 출구노드 $B_{p1}(i,j)$이며, i=0이면, i=β_R, j=0이면, j=β_C이다.(단, 마지막 블록은 i=m%β_R, j=n%β_C)

증명. $Q_4 \rightarrow Q_2$로 블록이 이동한다고 가정하자. 블록 엔트리집합 Q_4={B(i, j)|0≤i≤β_R, 0≤j≤β_C}, Q_2={B(i, j)|0≤i≤β_R, -β_C≤j-β_C≤0}이다. B(a, 0)(단, 0≤a≤β_R)이 Q_4의 출구노드라면, [ALG 2]의 (3)에 의해 Q_2'={B(i, j)|0≤i≤a, -β_C≤j-β_C≤0}만이 Q_4 엔트리값에 기여한다. 따라서, Q_2'의 입구노드는 B(a, β_C)이 된다. 마지막 블록의 경우, m%β_R=0, n%β_C=0이면, 행과 열 크기 i와 j의 최대값은 β_R과 β_C이며, 그렇지 않으면, i=m%β_R, j=n%β_C이므로, 입구노드는 i이다. □

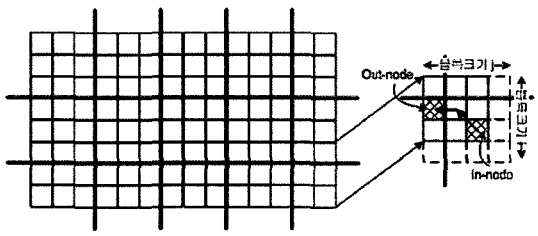

그림 7 Phase 1의 In/Out-node와 블록이동

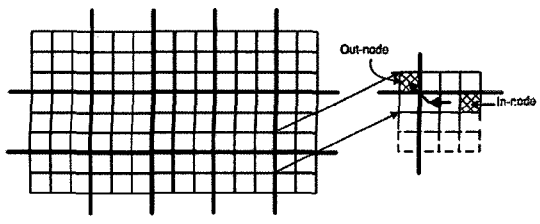

그림 8 Phase 2의 In/Out-node와 블록이동

주어진 문제를 분할시 정확하게 d(단, d>0정수)로 나눠지지 않을 때, In-node와 Out-node는 그림 7처럼, B(i, j)에서 i=2, j=2가 된다. 왼쪽 그림의 우측하단, 빗금 부분은 실제로 마지막 블록값을 계산하기위해 필요한 엔트리들이며, 오른쪽 그림은 블록(B)를 나타낸다. 블록에서 실선으로 된 부분만이 DPA가 실행되는 부분이다. 역추적과정에서 Q_4의 Out-node가 Q_2의 B(1,3)로 들어가면, Q_2의 In-node는 B(1,3)이 된다.

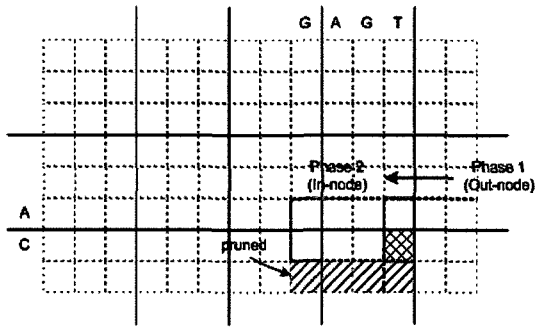

그림 9 Phase이동과 입/출구노드의 연결

그림 9는 마지막 블록(B_{p1})에서 중간블록(B_{p2})로의 블록이동에 따른 입/출구 노드의 연결을 보여준다. 블록크기 β(실선)는 4*4로 고정되지만, 블록 내에서 구해야하는 엔트리는 B_{p1}의 경우, 3*3이고, B_{p2}의 경우, 2*4이 되어, 블록 간 In-node와 Out-node연결을 통해 전체 최적정렬 계산에서 불필요한 부분이 삭제된다.

본 논문에서 서열의 전체 최적정렬을 얻기 위한 역추적과정 알고리즘은 다음과 같다.

[ALG 9] 블록 DPA시행 및 최적정렬 추출

(1) Set : In-node position y, x

i=m, j=n

(2) **while** i≥0 and j≥0

begin

(3) Initialization :

B'[0,..,y][0]←R_table[i-y,...,i][D_row_j]

B'[0][0,...,x]←C_table[D_col_i][j-x,...,j]

(4) Reconstruct :

ALG 2(y, x, S', T', C'): **(2)~(5)**

(5) Concatenate :

C←C||C'

(6) Block Shift :

if y=0 *and* x=0 **then** i=i-β_R, j=j-β_C

else if y=0 and x≠0 **then** i=i-β_R

else if y≠0 and x=0 **then** j=j-β_C

(7) **if** i=0 or j=0 **then** break

end

(8) Give output: C

전체 최적정렬을 위해 분할수 d로부터 생성된 부분블록들 중에서 마지막 블록부터 최초 블럭까지 순차적으로 연결되는 블록들에 대해 DPA를 시행한다. [알고리즘 9]의 (4)로부터 블록 내 부분최적정렬을 구하고, 다음으로 계산할 블록을 선택하기위해 (6)에서 현재 블록에서 *out-node*의 위치가 0일 때를 고려한다. 좌선에 위치(x=0)할 때, Q_2로, 상선에 위치(y=0)할 때, Q_3으로, 첫 번째 엔트리에 위치(x=0, y=0)할 때, Q_1블록으로 이동하고, 이동된 블록에서의 in-node는 이전 블록의 out-node가 된다. 블록 선택이 끝나면, 다시 in-node에 해당하는 블록의 엔트리 값들을 R_table과 C_table로부터 가져와 DPA를 시행한다. 이러한 과정을 비교하는 서열의 첫 번째 서열까지 진행한다.

[ALG 8]과 [ALG 9]가 수행되는 전체 과정을 모식도로 표현하면 다음과 같다.

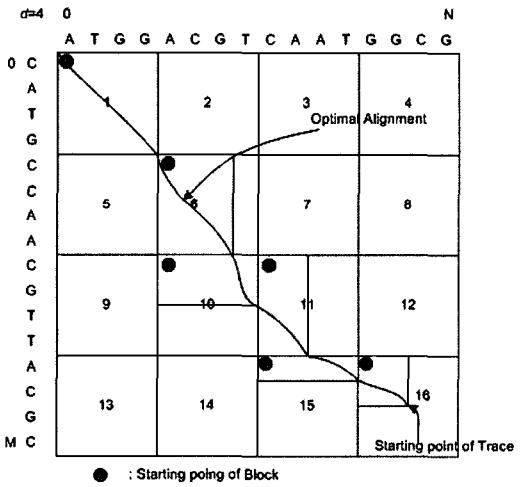

그림 10 분할기반 선형공간 ALG의 추적과정

그림 10에서 역추적과정이 시작되는 곳은 #16이며, • (dot)이 블록이동을 통하여 단어 단위로 비교가 시작되는 지점이다. 실선은 전체 최적정렬이며, 이는 블록 내 최소한의 엔트리들만을 계산하며 구해진다. 두 서열의 첫 번째 문자까지 블록계산이 되는 경로는, #16→#15→#11→#10→#6→#1이다.

기존 DPA에서는 최적정렬을 구하기 위해 비교하고자 하는 심벌 이후의 모든 심벌에 대해 엔트리 값을 계산하지만, 제안하는 알고리즘에서는 단어단위로 엔트리 값을 계산하게 된다. 이때 필요한 추가적 공간은 블록크기 ($\beta_R * \beta_C$) 만큼의 공간만 필요하게 된다.

4.2 알고리즘의 공간분석

제안하는 알고리즘의 공간 복잡도는 비교하고자하는 두 서열 S, T의 길이를 m과 n, 분할수를 정수 d라 할 때, 다음과 같다.

분할테이블(C_table, R_table)을 위해, $(m+n)*(d+1)$을 요구하고, 중간테이블(H Table)을 위해 $(m+1)*2$를 필요로 한다. 블록테이블의 경우, [정리 2]에 의해 최대 $(m/d+1)*(n/d+1)$이다. 즉, 총 공간비용은 $(m+n)(d+1)+(m+1)*2+(m*n)/d^2+(m+n)/d+1$ 이다. 이는 축약하면, $(m+n)d+(m*n)/d^2+(m+n)/d$이며, d를 크게 하면(단, $1 \le d \le m,n$), $(m+n)*d$에 가깝게 되어 $O(m+n)$의 범위를 만족한다. d를 1로 하였을 경우, DPA와 동일한 $O(m*n)$의 공간복잡도를 갖지만, 본 알고리즘은 $d \ge 2$인 경우를 고려하므로, 최소한 DPA보다 적은 공간을 사용한다.

4.3 알고리즘의 시간분석

본 알고리즘의 시간비용은 분할테이블 작성을 위해 필요한 $m*n$, 블록계산을 위해 $C*(m*n)/d^2$의(단, C=블록수) 시간을 소모한다. 이때 계산해야하는 블록크기는 최악의 경우, 블록 전체를 계산해야 하므로, $(m*n)/d^2$이 되며, 최선의 경우, 블록의 가장 작은 라인만 요구하는 경우로, $C*(m*n/d^2*\beta)$, (단, β는 블록크기)이다. [정리 2]에 의해 $\beta=m/d$이므로, $C*(n/d)$의 추가시간을 요구한다. 블록 수는 최선의 경우, d이며, 최악의 경우, $2d-1$이 된다. 즉, 전체 시간비용은 최악의 경우, $(m*n)+C(m*n)/d^2$이 되며, d를 m또는 n보다 작은 범위 안에서 크게 할 경우, 시간비용은 $m*n$에 가까워지게 된다. 즉, $O(m*n)$의 알고리즘이 된다.

5. 실 험

5.1 실험환경

본 논문에서는 Needleman의 DPA와 Hirschberg의 LSA, 제안한 알고리즘(d-LSA)을 일반적인 소프트웨어와 하드웨어를 사용하여 비교한다.

-Hardware : Intel Pentium Processor 1.73GHz, 1G RAM.

-Operating System : Windows XP Professional.

-구현도구 : Microsoft C++.

테스트를 위한 데이타는 잘 알려진 NCBI(National Center for Biotechnology Information)의 Entrez 검색 시스템에서 질의어 "nucleotide"를 통해 각각 100, 200, 400, 800, 1600, 3200, 6400, 12800개의 데이타를 추출하였다. (오차범위 : ±5%). 테스트의 처리시간은 각 샘플에 대해 10회 시행하고 평균시간을 대표 값으로 하였다.

5.2 실험결과

그림 11은 길이 800 미만의 서열에 대해 각각 DPA, LSA, d-LSA(d=10)의 방법으로 최적정렬을 찾을 때, 처리시간결과를 보여준다. 길이 200미만의 데이타는 산출시간이 0초로서 방법들 간의 구분이 안되지만, 서열의

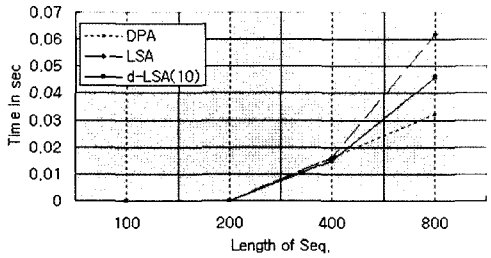

그림 11 짧은 서열에 대한 수행시간 비교

길이가 길어질수록 실선으로 표시된 d-LSA의 처리시간이 Hirschberg의 LSA보다 빠르게 수행된다. 대략, DPA 요구 시간과 LSA 요구 시간의 중간에 위치함을 알 수 있다. 본 논문에서는 서열의 길이가 더욱 긴 경우에 대해서는 DPA가 공간문제로 인해 적용이 불가능하므로, 긴 서열에 대해서는 LSA와 d-LSA만을 비교한다.

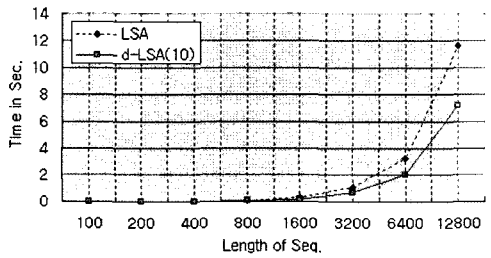

그림 12 긴 서열에 대한 수행시간 비교

그림 12는 서열의 길이를 12,800까지 확대하여 LSA와 d-LSA를 통하여 얻은 결과로서, 서열크기가 커지더라도 수행속도에 있어서는 여전히 LSA보다 빠름을 알 수 있다. 결과데이타는 정확히 12,800개 서열에 대해 LSA가 11.6초를, d-LSA는 7.6초를 소요한다.

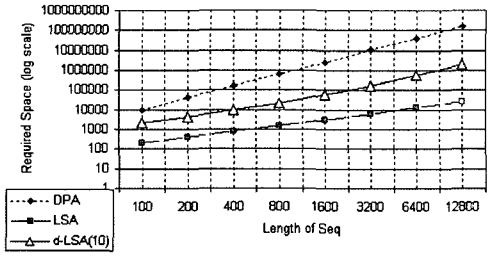

그림 13 공간요구 분석(DPA, LSA, d-LSA)

그림 13은 각 엔트리 당 1Byte의 공간을 필요로 한다고 하였을 때, 서열 길이에 따른 요구공간을 로그스케일로 표현한 것이다. 제안한 방법이 LSA보다 공간은 더 요구하지만, DPA보다는 훨씬 적은 공간을 요구한다. 이를 표로 나타내면 다음의 표 1과 같다.

표 1 공간요구 분석 테이블

Unit: Mega Byte

Length / Methods	100	200	400	800	1600	3200	6400	12800
DPA	0.01	0.04	0.16	0.64	2.56	10.24	40.96	163.00
LSA	0.0002	0.0004	0.0008	0.0016	0.0032	0.0064	0.0128	0.0256
d-LSA(10)	0.0021	0.0044	0.0096	0.0224	0.0576	0.1664	0.53	1.89

즉, 12,800개의 서열을 비교할 때, DPA가 163M Byte를 필요로 하는데 비해, d-LSA는 1.89M만 갖고, 데이타를 처리할 수 있다. 하지만, 처리시간에 있어서는 그림 11과 그림 12에서처럼 DPA와 가까운 시간으로 처리됨을 알 수 있다.

6. 결 론

서열의 최적정렬을 구하는 문제에 있어서 기존의 동적프로그래밍 알고리즘은 O(m*n)의 공간 및 시간 복잡도를 갖기 때문에 서열이 커질 경우, 컴퓨터 상에서 문제를 해결할 수 없게 된다. 이 경우, 실제 제한요소로 작용하는 공간문제는 Hirschberg에 의해 해결되었지만, 이는 처리시간이 DPA보다 2배정도 크기 때문에 공간은 더 사용하더라도 처리시간을 줄일 수 있는 방법이 필요하다. 따라서, 본 논문에서는 전체 매트릭스에 대해 분할기법을 적용함으로서 공간은 Hirschberg보다 많이 요구하지만, 처리시간은 빠른 알고리즘을 제안하였다.

제안한 알고리즘은 분할 기법을 통하여 O(m*n) 시간 복잡도를 갖게 된다. 좀 더 정확히 m*n에 가까운 시간을 필요로 한다. 이는 DPA가 갖는 요구시간에 근접한 시간이 되며, Hirschberg 알고리즘이 2(m*n)의 시간에 가까운 시간 비용을 요구하는 것을 고려할 때, 상당히 짧은 시간이 된다. 단, 분할로 인해 발생하는 공간 복잡도의 증가에 대해서는 d(m*n)이지만, 이는 충분히 허용될 수 있는 범위에 있음을 알 수 있다. 실제로 NCBI의 Database로부터 추출한 데이타에 대하여 서열의 크기를 달리하여 실험한 결과 제안한 방법이 충분히 허용될 수 있는 범위내의 공간을 사용하면서, Hirschberg보다 빠르게 데이타를 처리함을 확인할 수 있었다. 예를 들어, 1만개 정도의 서열에 대해 요구되는 공간은 1.2M정도만을 요구하게 된다(cf. DPA-100M, LSA-20K).

본 논문에서는 분할기법을 사용하여 알고리즘을 설계할 때, 정확히 분할되지 않는 서열에 대해 알고리즘을 일반화시켰다. 또한, 입구노드와 출구노드의 개념을 통하여 계산할 블록의 크기를 고정시키고, 역추적 과정 시 불필요한 엔트리에 대해 계산을 하지 않음으로서 실행시간을 단축하도록 알고리즘을 설계하였다.

향후 연구과제로 분할 수에 따른 처리시간과 요구 공간에 대한 상관관계를 분석하여 최적의 분할수를 찾아냄으로서 알고리즘의 효율성을 높일 수 있는 방법을 고

안할 필요성이 있다.

참 고 문 헌

[1] Setubal, J. and Meidanis, J. "Introduction to Computational Molecular Biology," *PWS Publishing Company*, 1997.

[2] Wagner, R. A. and Fischer, M. J. "The string-to-string correction problem," *J. ACM 21*, 168-173, Jan. 1974.

[3] Tao Jiang, Ying Xu and Michael Q. Zhang, "Current topics in computational molecular biology," *The MIT Press, 5, Cambridge Center, Cambridge*, 2002.

[4] Andreas D. Baxevanis and B. F. Francis Ouellette. "Bioinformatics : A practical guide to the analysis of genes and progeins," *2-ed; Willey & Sons*, 2001.

[5] Dennis A. Benson, Ilene Karsch-Mizrachi, David J. Lipman, James Ostell and David L. Wheeler. "GeneBank," *Nucleic Acids Research*, Vol. 24, Database issue, 2006.

[6] Cohen, J. "Bioinformatics: An introduction for computer scientists," ACM Comput. Surv. Vol. 36, No. 2, 122-158., Jun. 2004.

[7] Schwartz, S. Zhang, Z., Frazer, K.A., et al, "PipMaker-A Web Server for Aligning Two Genomic DNA Sequences," *Genome Research* 10, 577-686., 2002.

[8] Ning, Z., et al, "SSAHA: a fast search method for large DNA databases," *Genome Research 11*(10), 1725-1729., 2001.

[9] Delcher, A.L., Kasif, S., Fleishmann, R.D., Peterson, J., White, O. & Salzberg, S.L. "Alignment of whole genomes," *Nucleic Acids Research* 27, 2369-2376., 1999.

[10] Brudno, M. & Morgenstern, B. "Fast and sensitive alignment of large genomic sequences," in *'IEEE Computer Society Bioinformamtics Conference 2002'*, Standford University, CA, USA, pp. 138-147., 2002.

[11] Needleman, S. B. and Wunsch C.D. "A general method applicable to the search for similarities in the amino acid sequences of two proteins," *J. Mol. Biol.* vol. 48: 443-453, 1970.

[12] Hirschberg, D. S. "A linear space algorithm for computing maximal common subsequenes," *Commun. Assoc. Comput.* March. vol. 18: 341-343, 1975.

[13] Myers, E. W. and Miller, W. "Optimal alignments in linear space," *Comput. Applic. Biosci.* vol. 4: 11-17, 1988.

[14] Steven L. Salzberg. "Gene Discovery in DNA Sequences," *IEEE. Intelligent System.* Nov. 1999.

[15] F. S. Roberts. "Applied Combinatorics," *Prentice-Hall*, 1984.

[16] Arthur M. Lesk. "Introduction to Bioinformatics," *Oxford University Press Inc*, 2002.

안 희 국

1991년~1998년 강원대학교 미생물학과 졸업(학사). 2001년~2003년 강원대학교 컴퓨터과학과 졸업(석사). 2003년~2007년 강원대학교 컴퓨터과학과 졸업(박사). 2006년~현재 송호대학 겸임강사. 관심분야는 생물정보학, 텍스트마이닝, 알고리즘

노 희 영

1964년~1972년 고려대학교 독문학과 졸업 1974년~1982년 독일 Dortmund 공대 Informatik(전자계산학). 1983년~1984년 대덕 연구단지 한국표준연구소 전산실 선임연구원. 1984년 2월~현재 강원대학교 IT특성화학부대학 컴퓨터과학과 교수. 관심분야는 컴파일러, 자연어처리, 소프트웨어공학