DOI: 10.3745/KIPSTA.2009.16A.5.319

문자열의 최장 공통 부분문자열과 최대 반복자를 구하기 위한 상수시간 RMESH 알고리즘

한 선 미[†] · 우 진 운[††]

요 약

문자열 연산이 계산 생물학 분야에 응용되면서 효율적인 문자열 연산을 위한 다양한 자료구조와 알고리즘이 연구되고 있다. 최장 공통 부분문자열 문제는 두 개 이상의 문자열에서 가장 길게 일치하는 부분문자열을 찾는 연산이며, 최대 반복자 문제는 하나의 문자열에서 두 번 이상 반복되는 부분문자열을 찾는 연산이다. 이 연산은 패턴 매칭, 유사도 측정 등의 문자열 처리 분야에서 중요하게 사용되고 있다.

본 논문에서는 RMESH(Reconfigurable MESH) 구조에서 3-차원 $n \times n \times n$ 프로세서를 사용하여 두 문자열의 최장 공통 부분문자열을 구하는 알고리즘과 주어진 문자열의 최대 반복자를 찾는 알고리즘을 제안하며, 이 알고리즘들은 모두 $O(1)$ 시간 복잡도를 갖는다.

키워드 : 문자열, 최장 공통 부분문자열, 최대 반복자, RMESH

Constant Time RMESH Algorithm for Computing Longest Common Substring and Maximal Repeat of String

Seon Mi Han[†] · Jin Woon Woo[††]

ABSTRACT

Since string operations were applied to computational biology area, various data structures and algorithms for computing efficient string operations have been studied. The longest common substring problem is an operation to find the longest matching substring in more than two strings, and maximal repeat of string problem is an operation to find substrings repeated more than once in the given string. These operations are importantly used in the string processing area such as pattern matching and likelihood measurement.

In this paper, we present algorithms to compute the longest common substring of two strings and to find the maximal repeat of string using three-dimensional $n \times n \times n$ processors on RMESH(Reconfigurable MESH). Our algorithms have $O(1)$ time complexity.

Keywords : String, Longest Common Substring, Maximal Repeat, RMESH

1. 서 론

최근 문자열 연산들이 계산 생물학 분야에 응용되면서 효율적인 문자열 연산을 위한 자료구조와 알고리즘들이 연구되고 있다. 이러한 문자열 연산에는 문자열 패턴 매칭(string pattern matching), 최장 공통 부분문자열(longest common substring) 찾기, 최대 반복자(maximal repeat) 찾기 등이 있다.

최장 공통 부분문자열은 두 개 이상의 문자열에서 가장 길게 일치하는 부분문자열을 의미한다[1, 2]. 최장 공통 부분문자열을 찾는 문제는 패턴 매칭, 유사도 측정 등의 문자열 처리 분야에서 중요하게 사용될 뿐만 아니라, DNA 순서 간 일치성 여부를 찾아내는 등의 생물정보학 분야에서도 아주 중요하다[3].

최장 공통 부분문자열을 구하는 단일 프로세서 알고리즘은 주로 동적 프로그래밍을 이용하거나 접미사 트리를 이용한다. 먼저, 동적 프로그래밍은 두 문자열의 편집 거리(edit distance) 계산을 통한 방식으로 최장 공통 부분문자열을 계산한다[1, 4]. 이는 두 문자열의 길이를 각각 m 과 n 일 때 O(mn) 시간복잡도를 가진다. 접미사 트리를 이용하는 방법은 선형 시간복잡도를 갖지만 많은 공간을 사용하는 단점이

※ 이 연구는 2008학년도 단국대학교 대학연구비의 지원으로 연구되었음.
† 준 회 원 : 단국대학교 컴퓨터과학전공 박사과정
†† 종신회원 : 단국대학교 컴퓨터학부 교수(교신저자)
논문접수 : 2009년 1월 22일
수 정 일 : 1차 2009년 6월 2일
심사완료 : 2009년 6월 3일

있다[2]. 접미사 트리 대신 접미사 배열을 사용하여 공간을 적게 사용하는 알고리즘도 제안되었다[5].

최대 반복자는 하나의 문자열에서 두 번 이상 반복되는 부분문자열을 의미하며, 최대 반복자 찾기 문제는 주어진 문자열에 존재하는 모든 최대 반복자들을 찾는 것이다. 최장 공통 부분문자열 문제와 유사하게 접미사 트리를 이용하여 O(n) 시간 복잡도 알고리즘이 알려져 있다[2]. 또한 여러 문자열에서 최장 공통 반복자를 선형시간에 찾는 알고리즘도 알려져 있다[6].

문자열 연산과 관련된 병렬 알고리즘이 많이 연구되어 있다[7-12]. 대부분 CRCW-PRAM 모델에서 동작하는 알고리즘들로서 접미사 트리를 생성하여 문자열 매칭과 최장 반복 부분문자열을 O(log n) 시간에 수행하거나[7], 상수 시간에 문자열 매칭을 수행하는 알고리즘을 제안한다[10].

그러나, RMESH 모델이 제안된 이래, 영상처리 분야에서는 많은 RMESH 알고리즘들이 발표되었으나, 문자열 처리 분야에는 활발하게 발표되지 못하고 있다. Lee와 Ercal이 RMESH 모델에서 상수시간 문자열 매칭 알고리즘을 제안하였고[11], Datta와 Subbiah는 RMESH 모델에서 문자열 처리를 위한 상수시간 알고리즘들을 제안하였다[12].

본 논문은 RMESH 모델에서 최장 공통 부분문자열과 최대 반복자를 찾는 상수 시간 알고리즘들을 제안한다. 최장 공통 부분문자열 알고리즘은 동적 프로그래밍 기법을 사용하며 RMESH 모델의 각 프록세서의 스위치를 효율적으로 작동시킴으로써 상수 시간에 동작한다. 최대 반복자 알고리즘은 최장 공통 부분문자열 알고리즘을 적절히 이용함으로써 상수 시간에 동작한다.

본 논문과 [11, 12]에서는 두 문자열을 비교하기 위해 동적 프로그래밍 기법을 사용하고 이를 버스로 연결하는 점에서는 유사하다. 그러나 [11]은 2-차원 RMESH를 사용하여 상수시간에 문자열 매칭과 근사 문자열 매칭을 수행하는 알고리즘을 제안한다. 본 논문과 [12]는 3-차원 RMESH를 사용하여 최장 공통 부분문자열과 최대 반복자를 상수시간에 구하는 알고리즘을 제안한다. 이 문제를 해결하는 과정에서 [12]는 최대값을 구하는 일반적인 RMESH 알고리즘을 사용하나, 본 논문은 연결된 버스에서 직접 최장 공통 부분문자열과 최대 반복자를 구하기 위한 효율적인 기본 연산들을 사용한다.

2. RMESH 구조

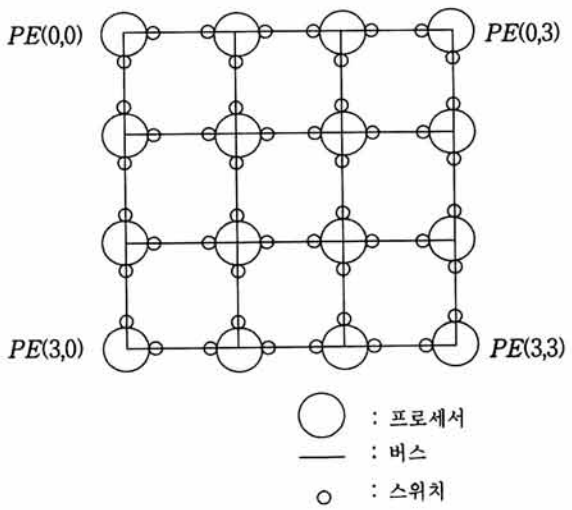
RMESH는 Reconfigurable MESH의 약어로서 기존의 메쉬(mesh) 구조에 동적으로 재구성 가능한 버스 시스템을 결합한 구조로서 Miller, Prasanna-Kumar, Reisis, Stout에 의하여 제안되었으며[13], 구조적인 장점 때문에 다양한 분야에서 연구되었고 효율적인 알고리즘들이 개발되었다[14, 15]. 또한 버스 시스템의 재구성 방법 면에서 서로 차이를 갖는 PARBUS 구조와 MRN 구조가 제안되었다[16, 17].

2.1 2-차원 RMESH

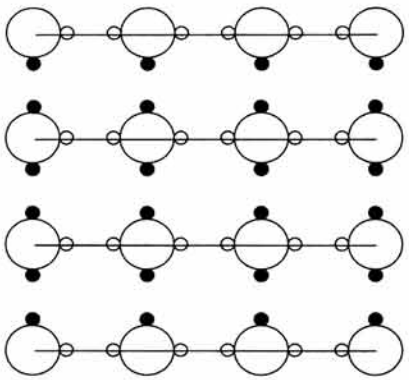
크기가 $n \times n$인 2-차원 RMESH의 기본 구조는 메쉬이며 프로세서들 사이의 통신을 위하여 브로드캐스트 버스(broadcast bus)가 존재한다. 예를 들어, (그림 1)은 4×4 RMESH 구조를 보여준다. 프로세서들을 식별하기 위해 각 프로세서에게 $PE(i,j)$ 를 부여한다. 이때 $0 \leq i,j < n$, i는 행의 인덱스이고, j는 열의 인덱스이다.

브로드캐스트 버스상의 통신 제어를 위하여 버스 스위치가 있다. 버스 스위치들은 각 프로세서의 상, 하, 좌, 우에 하나씩 존재하는데, 이를 각각 N(north), S(south), W(west), E(east)라 한다. 버스 스위치는 각 프로세서의 소프트웨어에 의하여 $O(1)$ 시간에 조작되며, 스위치의 개폐 여부에 따라 브로드캐스트 버스를 다수의 서브버스(subbus)들로 재구성이 가능하다. 예를 들어, 각 프로세서가 자신의 S와 N 스위치를 끊고 E와 W 스위치를 연결한다면 여러 개의 서브버스가 형성되는데, 이를 (그림 2)(a)와 같은 행 버스(row bus)라 하고, 자신의 E와 W 스위치를 끊고 S와 N 스위치를 연결한다면 여러 개의 서브버스가 형성되는데, 이를 (그림 2)(b)와 같은 열 버스(column bus)라 한다. 또한 각 프로세서가 외부적으로 S와 N 스위치, E와 W 스위치를 연결하고, 내부적으로 N과 E 스위치, S와 W 스위치를 연결하게 되면 여러 개의 서브버스가 형성되며 이를 (그림 2)(c)와 같은 대각선 버스(diagonal bus)라 한다.

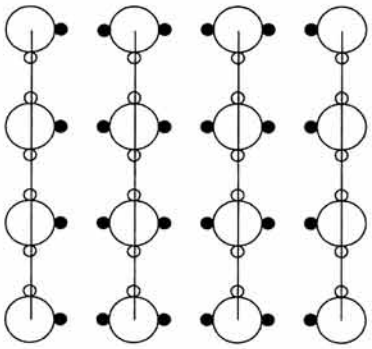
두 개의 프로세서들은 충돌이 없는 한 공통된 하나의 특정 스위치를 동시에 개폐할 수 있다. 버스상에는 특정 시간에 단 하나의 프로세서만이 데이터를 실을 수 있으며, 서브버스 위에 실린 데이터는 단위 시간에 그 버스에 연결된 모든 프로세서에게 전달될 수 있다. 만약 한 프로세서가 서브버스상에 있는 모든 프로세서에게 레지스터(register) X의 값을 브로드캐스트하려면 broadcast(X) 명령을 사용하고, 브로드캐스트 버스의 내용을 읽어 레지스터 R에 저장하려면 R := content(broadcast bus) 명령을 사용한다. 따라서 데이



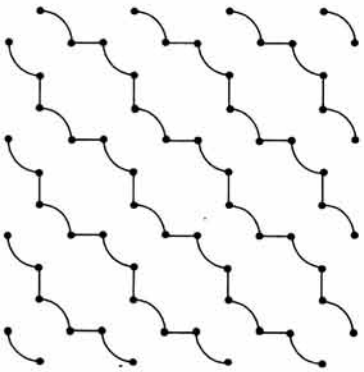
(그림 1) 4×4 RMESH 구조



(a) 행 버스



(b) 열 버스



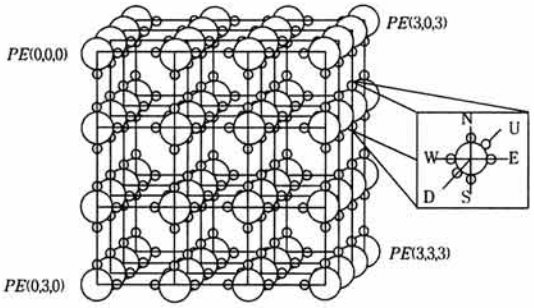
(c) 대각선 버스

(그림 2) 행, 열, 대각선 서브버스

터 브로드캐스트는 $O(1)$ 시간에 수행된다.

2.2 3-차원 RMESH

2-차원 RMESH를 확장하여 3-차원 RMESH를 구성할 수 있다. 3-차원 RMESH에서는 각 프로세서에게 $PE(l,i,j)$를 부여한다. 이때 $0 \leq l,i,j < n$, l은 각 프로세서가 위치한 계층(layer)이고, i와 j는 계층 l에서의 행과 열의 인덱스이다. 예를 들어, (그림 3)은 4×4×4 RMESH를 보여준다. 버스 스위치들은 기본적으로 2-차원 RMESH와 같이 N, S, W, E 스위치가 존재하며, 추가적으로 각 프로세서마다 계층



(그림 3) 4×4×4 RMESH 구조

을 연결하는 U(up)와 D(down) 스위치가 존재한다. 그리고 모든 프로세서의 N, S, W, E 스위치를 끊고 U와 D 스위치를 연결하면 여러 개의 서브버스가 형성되는데, 이를 UD 버스라 한다.

3. 최장 공통 부분문자열 구하기

RMESH 구조에서 두 문자열의 최장 공통 부분문자열을 구하기 위해서는 동적 프로그래밍 기법을 적절히 이용한다. 동적 프로그래밍은 두 문자열의 편집 거리(edit distance) 계산을 통한 방식으로 공통 부분문자열을 계산한다[1, 4]. 이차원 테이블에서 각 테이블 원소의 행과 열에 해당하는 문자가 일치하면 1, 그렇지 않으면 0으로 표시한 후에 대각선 방향으로 연속된 1의 개수가 편집 거리에 해당한다. 편집 거리가 가장 긴 부분문자열이 최장 공통 부분문자열에 해당한다.

예를 들어, A = "abcabbac", B = "acabbca" 라 할 때, 문자열의 길이가 각각 |A| = 8, |B| = 7 이므로 문자열 B에 특수 문자 '$'를 추가한다(이때 특수 문자 '$'는 두 문자열에 포함되는 않는 문자로 선택한다). 문자열 A는 테이블의 위쪽, 문자열 B는 테이블의 왼쪽에 두고서 행과 열이 교차하는 테이블의 원소에 1 또는 0을 표시한다. (그림 4)(a)는 행과 열의 문자를 서로 비교하여 같으면 1, 같지 않으면 0으로 표시한 것이다. (그림 4)(b)는 (그림 4)(a)를 바탕으로 편집 거리를 계산하였으며, 가장 긴 편집 거리를 가진 부분문자열이 최장 공통 부분문자열에 해당한다. 이 예에서는 "cabb"가 최장 공통 부분문자열이다.

	a	b	c	a	b	b	a	c
a	1	0	0	1	0	0	1	0
c	0	0	1	0	0	0	0	1
a	1	0	0	1	0	0	1	0
b	0	1	0	0	1	1	0	0
b	0	1	0	0	1	1	0	0
c	0	0	1	0	0	0	0	1
a	1	0	0	1	0	0	1	0
$	0	0	0	0	0	0	0	0

(a) 행과 열 문자의 일치성을 0과 1로 표시

	a	b	c	a	b	b	a	c
a	1	0	0	1	0	0	1	0
c	0	0	1	0	0	0	0	2
a	1	0	0	2	0	0	1	0
b	0	2	0	0	3	1	0	0
b	0	1	0	0	1	4	0	0
c	0	0	2	0	0	0	0	1
a	1	0	0	3	0	0	1	0
$	0	0	0	0	0	0	0	0

(b) 편집 거리

(그림 4) 최장 공통 부분문자열을 구하는 예

두 문자열의 길이가 각각 m, n ($m \le n$) 일 때, $n \times n \times n$ 3차원 RMESH 구조에서 최장 공통 부분문자열을 구하는 알고리즘을 요약하면 알고리즘 1과 같이 6 단계로 구성된다.

초기에 길이가 짧은 문자열에 대해 $n - m$ 개의 특수문자가 뒤에서부터 채워져 있으며, 두 문자열은 $n \times n \times n$ RMESH의 계층 0에 속하는 프로세서들의 첫 번째 행과 열의 프로세서에 저장되어 있다고 가정한다. 즉, 계층 0의 프로세서 $PE(0,0,j)$ $(0 \le j \le n-1)$는 A 문자열의 한 문자씩을 가지고, 계층 0의 프로세서 $PE(0,i,0)$ $(0 \le i \le n-1)$는 B 문자열의 한 문자씩을 가진다.

[알고리즘 1] 최장 공통 부분문자열을 계산하는 RMESH 알고리즘

[단계 1]: 계층 0의 첫 번째 행에 속하는 프로세서 $PE(0,0,j)$ $(0 \le j \le n-1)$는 자신의 문자를 열 버스를 이용하여 브로드캐스트하고, 계층 0의 첫 번째 열에 속하는 프로세서 $PE(0,i,0)$ $(0 \le i \le n-1)$는 자신의 문자를 행 버스를 이용하여 브로드캐스트한다.

[단계 2]: 계층 0의 모든 프로세서 $PE(0,i,j)$는 행과 열 버스로 전달받은 두 문자를 서로 비교하여 같으면 대각선 버스를 형성하고, 같지 않으면 대각선 버스를 차단한다.

[단계 3]: 대각선 버스를 이용하여 프로세서들을 세그먼트로 분리하며, 세그먼트의 첫 프로세서는 *Start* 레지스터를 1로, 마지막 프로세서는 *End* 레지스터를 1로 설정한다.

[단계 4]: *Start* = 1 인 프로세서는 자신의 프로세서 *id* 값을 대각선 버스를 통하여 브로드캐스트한다.

[단계 5]: *End* = 1 인 프로세서는 전달받은 프로세서의 *id* 값을 이용하여 *(Distance, i, j)* 값을 저장한다.

[단계 6]: *(Distance, i, j)*의 *Distance* 값을 기준으로 하여 내림차순으로 정렬한다. 이때 계층 0의 프로세서 $PE(0,0,0)$가 가진 *(Distance, i, j)*의 값이 최장 공통 부분문자열을 길이와 위치를 나타낸다.

최장 공통 부분문자열을 계산하는 RMESH 알고리즘이 수행되는 과정을 단계별로 살펴보자.

[단계 1]에서 계층 0의 프로세서 $PE(0,0,j)$ $(0 \le j \le n-1)$는 자신의 문자를 열 버스를 이용하여 브로드캐스트하고, 계층 0의 프로세서 $PE(0,i,0)$ $(0 \le i \le n-1)$는 자신의 문자를 행 버스를 이용하여 브로드캐스트한다. 따라서 이 단계는 두 번의 브로드캐스트 만을 수행하므로 $O(1)$ 시간에 수행 가능하다.

[단계 2]에서 계층 0의 모든 프로세서는 행과 열 버스를 통해 전달받은 두 문자를 비교하여 같으면 대각선 버스를 형성하고, 같지 않으면 대각선 버스를 차단한다. 이 결과로 계층 0의 프로세서들이 대각선 방향으로 세그먼트를 형성하게 된다. 이 과정은 동적프로그래밍 기법에서 설명된 바와 같이 두 문자가 같으면 1, 같지 않으면 0으로 나타내는 것과 같다. 이 단계는 단순히 한 번의 문자 비교만 하게 되므로 $O(1)$ 시간에 수행 가능하다.

[단계 3]에서는 대각선 방향으로 세그먼트를 형성하게 되며, 대각선 버스에서 왼쪽이 차단되고 오른쪽이 연결된 프로세서는 *Start* 레지스터에 1을 저장하고, 그렇지 않으면 0을 저장한다. 여기서 *Start* 레지스터가 1인 프로세서는 하나의 세그먼트 시작을 의미한다. 왼쪽은 연결되었으나 오른쪽이 차단된 프로세서는 *End* 레지스터에 1을 저장하고, 그렇지 않으면 0을 저장한다. 여기서 *End* 레지스터가 1인 프로세서는 세그먼트의 마지막에 해당한다. 이 단계는 대각선 버스의 인접한 프로세서만을 확인하므로 $O(1)$ 시간에 수행 가능하다.

예를 들어, (그림 4)의 두 문자열에 대해 단계 3까지 적용하면 (그림 5)와 같이 나타낼 수 있다.

[단계 4]에서 *Start* = 1 인 프로세서는 세그먼트 내의 프

	$PE(0,0,0)$ a	$PE(0,0,1)$ b	$PE(0,0,2)$ c	$PE(0,0,3)$ a	$PE(0,0,4)$ b	$PE(0,0,5)$ b	$PE(0,0,6)$ a	$PE(0,0,7)$ c
$PE(0,0,0)$ a	Start=1 End=1	0	0	Start=1 End=1	0	0	Start=1	0
$PE(0,1,0)$ c	0	0	Start=1	0	0	0	0	End=1
$PE(0,2,0)$ a	Start=1	0	0		0	0	Start=1 End=1	0
$PE(0,3,0)$ b	0	End=1	0	0		Start=1 End=1	0	0
$PE(0,4,0)$ b	0	Start=1	0	0	Start=1 End=1	End=1	0	0
$PE(0,5,0)$ c	0	0		0	0	0	0	Start=1 End=1
$PE(0,6,0)$ a	Start=1 End=1	0	0	End=1	0	0	Start=1 End=1	0
$PE(0,7,0)$ $	0	0	0	0	0	0	0	0

(→ 대각선 버스)

(그림 5) 단계 1에서 단계 3까지의 적용 예

로세서들에게 자신의 프로세서 *id* 값을 대각선 버스를 통하여 브로드캐스트 하는데, 프로세서 *id* 는 프로세서의 행과 열 번호로 나타낸다. 즉, 프로세서 $PE(0,i,j)$의 경우, *(i, j)*를 브로드캐스트한다. 이 과정은 세그먼트를 나타내는 대각선 버스 상에서 한 번의 브로드캐스트만 일어나므로 $O(1)$ 시간에 수행된다.

[단계 5]에서 *End* = 1 인 프로세서는 전달받은 프로세서의 *id* 를 이용하여 *(Distance, i, j)* 값을 저장한다. 여기서 *Distance*는 편집 거리에 해당하며 자신의 열 번호에서 전달받은 프로세서의 *id* 의 j 값을 뺀 값이며, *i*와 *j*는 전달받은 프로세서 *id* 값에 해당한다. 즉, 프로세서 $PE(0,i',j')$가 *(i, j)* 값을 전달받았다면, *Distance* = $j'-j+1$ 로 계산되고, *(Distance, i, j)* 값을 저장한다. *End* = 0 인 프로세서는 (0, 0, 0)의 값을 저장한다. 이 단계에서는 세그먼트의 마지막 프로세서가 *(Distance, i, j)* 필드를 계산하는 시간만 소요하므로 걸리는 시간은 $O(1)$이다.

예를 들어, (그림 5)에서 *End* = 1 인 프로세서가 저장한 *(Distance, i, j)* 필드 값을 나타내면 (그림 6)과 같다.

[단계 6]은 *(Distance, i, j)*의 *Distance* 값을 기준으로 하여 내림차순으로 정렬한다. Nigam과 Sahni[17]는 rotate sort 알고리즘을 3-차원 $n\times n\times n$ RMESH에 적용하여 n^2 개의 데이터를 $O(1)$ 시간에 정렬할 수 있는 알고리즘을 제안하였는데, 이 알고리즘에서 초기의 n^2 개의 데이터는 계층 0에 속하는 $PE(0,i,j)$ $(0 \leq i,j < n)$에 존재하며, 정렬된 결과는 계층 0에 속하는 프로세서에 row-major 순서로 하나씩 저장된다. 즉, $PE(0,0,0)$, $PE(0,0,1)$, $\cdots$, $PE(0,1,0)$, $PE(0,1,1)$, $\cdots$, $PE(0,n-1,n-1)$의 순서로 저장된다. 이 단계에서 이 정렬 알고리즘을 이용할 때 *(Distance, i, j)* 값들을 $O(1)$에 정렬할 수 있다. 이때, 계층 0의 프로세서 $PE(0,0,0)$가 가진 *(Distance, i, j)*의 값이 최장 공통 부분문자열의 길이와 위치를 나타낸다.

예를 들어, (그림 6)에 대해 단계 6이 적용되면 $PE(0,0,0)$는 (4, 1, 2)의 값을 가지게 된다. 이것은 최장 공통 부분문자열의 길이가 4이며, 그 위치는 B = "acabbca" 에서 1-번째, A = "abcabbac" 에서 2-번째 문자부터 시작함을 의미한다(문자열은 0-번째 문자부터 시작한다). 따라서 최장 공통 부분문자열은 "cabb"가된다..

지금까지 알고리즘 1의 각 단계가 모두 $O(1)$ 시간에 수행될 수 있음을 설명하였으며, 정리 1과 같이 요약할 수 있다.

[정리 1] 두 문자열의 길이가 각각 m, n $(m \leq n)$ 일 때, $n\times n\times n$ 3차원 RMESH 구조에서 최장 공통 부분문자열을 구하는 알고리즘은 $O(1)$ 시간에 수행된다.

4. 최대 반복자 구하기

최대 반복자는 하나의 문자열에서 두 번 이상 반복되는 최대 부분문자열을 의미한다. 최대 반복자란 바로 인접한 문자를 포함할 때 반복자가 되지 않음을 뜻한다. 예를 들어, 하나의 문자열 "xabcyabcbc" 에서 두 번 이상 반복되는 부분문자열들을 구하면 "abc", "ab", "bc" 이다. "abc"의 경우 "xabcyabcbc" 와 "xabcyabcbc" 에서 반복되고 인접한 문자가 서로 다르므로 최대 반복자가 된다. "ab"의 경우 "xabcyabcbc" 와 "xabcyabcbc" 에서 반복되나 오른쪽에 인접한 문자 'c'가 같으므로 최대 반복자는 될 수 없다. "bc"의 경우, "xabcyabcbc", "xabcyabcbc", "xabcyabcbc" 에서 반복된다. "xabcyabcbc"와 "xabcyabcbc"에서는 최대 반복자가

	$PE(0,0,0)$ a	$PE(0,0,1)$ b	$PE(0,0,2)$ c	$PE(0,0,3)$ a	$PE(0,0,4)$ b	$PE(0,0,5)$ b	$PE(0,0,6)$ a	$PE(0,0,7)$ c
$PE(0,0,0)$ a	(1,0,0)	(0,0,0)	(0,0,0)	(1,0,0)	(0,0,0)	(0,0,0)		(0,0,0)
$PE(0,1,0)$ c	(0,0,0)	(0,0,0)		(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)	(2,0,6)
$PE(0,2,0)$ a		(0,0,0)	(0,0,0)		(0,0,0)	(0,0,0)	(1,2,6)	(0,0,0)
$PE(0,3,0)$ b	(0,0,0)	(2,3,1)	(0,0,0)	(0,0,0)		(1,3,5)	(0,0,0)	(0,0,0)
$PE(0,4,0)$ b	(0,0,0)		(0,0,0)	(0,0,0)	(1,4,4)	(4,1,2)	(0,0,0)	(0,0,0)
$PE(0,5,0)$ c	(0,0,0)	(0,0,0)		(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)	(1,5,7)
$PE(0,6,0)$ a	(1,6,0)	(0,0,0)	(0,0,0)	(3,6,3)	(0,0,0)	(0,0,0)	(1,6,6)	(0,0,0)
$PE(0,7,0)$ $	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)

(그림 6) 단계 5가 실행된 후의 (*Distance, i, j*) 값

될 수 없으나, "xabcyabcbc"와 "xabcyabcbc"에서 최대 반복자가 되므로 "bc"는 최대 반복자라 할 수 있다.

최대 반복자를 구하기 위해 두 문자열의 최장 공통 부분 문자열을 구하기 위한 동적 프로그래밍 기법을 적절히 이용할 수 있다. 주어진 문자열을 동일한 두 문자열로 생각하여 두 문자열의 편집 거리를 계산하고, 편집 거리가 2 이상이 되는 부분 문자열들이 최대 반복자 후보들이 된다. 최대 반복자 후보들 중에서 중복된 부분 문자열들 중 하나만을 남기고 나머지들을 제외시키면, 최종 남은 부분 문자열들이 최대 반복자가 된다.

위에서 사용한 문자열 "xabcyabcbc" 을 이용하여 예를 들어 보자. 두 문자열을 동일한 문자열 A = "xabcyabcbc", B = "xabcyabcbc"로 두고 편집 거리를 구하면 (그림 7)(b)와 같게 된다.

(그림 7)(b)에서 거리가 2 이상인 부분 문자열(단, 대각선이 나타내는 문자열은 제외)들은 최대 반복자의 후보들이다. 이때 대각선을 경계로 대칭을 이루므로 대각선 아래쪽 또는 위쪽 중 한 곳의 부분 문자열이 대상이 되며, (그림 7)(b)에서 "abc", "bc", "bc" 가 후보들이다. 행 번호와 편집거리가 같은 후보들 중 하나만을 남기고 나머지를 제외시키면 남은 부분문자열들이 최대 반복자가 되며, 이 예에서는 "abc"와 "bc"가 최대 반복자가 된다.

주어진 문자열의 길이가 n 일 때, $n\times n\times n$ 3차원 RMESH 구조에서 최대 반복자를 구하는 알고리즘을 요약하면 알고리즘 2와 같이 12 단계로 구성된다.

초기에 주어진 문자열은 $n\times n\times n$ RMESH의 계층 0에 속하는 프로세서들의 첫 번째 행과 열의 프로세서에 저장되어 있다고 가정한다. 즉, 계층 0의 프로세서 $PE(0,0,j)$ $(0\le j\le n-1)$와 프로세서 $PE(0,i,0)$ $(0\le i\le n-1)$는 주어진 문자열의 한 문자씩을 가진다.

[알고리즘 2] 최대 반복자들을 계산하는 RMESH 알고리즘

[단계 1]: 계층 0의 첫 번째 행에 속하는 프로세서 $PE(0,0,j)$ $(0\le j\le n-1)$는 자신의 문자를 열 버스를 이용하여 브로드캐스트하고, 계층 0의 첫 번째 열에 속하는 프로세서 $PE(0,i,0)$ $(0\le i\le n-1)$는 자신의 문자를 행 버스를 이용하여 브로드캐스트한다.

[단계 2]: 계층 0의 모든 프로세서 $PE(0,i,j)$는 행과 열 버스로 전달받은 두 문자를 서로 비교하여 같으면 대각선 버스를 형성하고, 같지 않으면 대각선 버스를 차단한다.

[단계 3]: 대각선 버스를 이용하여 프로세서들을 세그먼트로 분리하며, 세그먼트의 첫 프로세서는 *Start* 레지스터를 1로, 마지막 프로세서는 *End* 레지스터를 1로 설정한다.

[단계 4]: *Start* = 1 인 프로세서는 자신의 프로세서 *id* 값을 대각선 버스를 통하여 브로드캐스트한다.

[단계 5]: *End* = 1 인 프로세서는 전달받은 프로세서의 *id* 값을 이용하여 편집 거리인 *Distance* 를 계산하여, *(i, j, Distance)* 값을 저장한다. 이때 i 는 전달받은 프로세서의 행 번호이고, j 는 열 번호이다.

[단계 6]: <*i, Distance, j*> 크기의 오름차순으로 정렬한다.

[단계 7]: 계층 0의 행 i에 있는 값들을 계층 i의 행 0으로 이동시킨 후, 홀수번째 계층에 있는 값들을 역순으로 permute한다. 그리고 $n\times n\times n$ RMESH를 $n\times n^2$ RMESH의 개념을 가진 구조가 되도록 재구성한다.

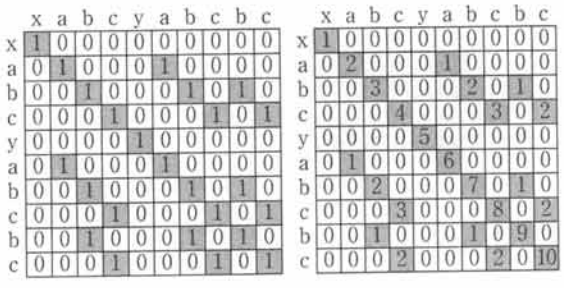
[단계 8]: 행 번호 i 와 편집거리 *Distance*가 같은 프로세서들을 세그먼트로 분리하며, 세그먼트의 첫 프로세서는 *Live = 1* 을 추가하고, 세그먼트의 나머지 프로세서들은 *Live = 0* 을 추가하여 *(i, j, Distance, Live)* 를 저장한다.

[단계 9]: *(i, j, Distance, Live)* 값을 계층 0의 프로세서에 row-major 순서로 하나씩 저장한다.

[단계 10]: *(i, j, Distance, Live)* 값을 계층 0의 프로세서 $PE(0,i,j)$로 이동한다.

[단계 11]: *(i, j, Distance, Live)* 값을 계층 0에서 대각선에 대해 대칭인 프로세서 $PE(0,j,i)$로 이동한다.

[단계 12]: 두 값의 Live 값이 모두 1 인 경우에 *(i, j, Distance)* 값을 저장하고, 그렇지 않으면 이 값을 제거한다.



(a) 행과 열 문자의 일치성을 0과 1로 표시 (b) 편집 거리

(그림 7) 최대 반복자를 구하는 예

최대 반복자를 계산하는 RMESH 알고리즘이 수행되는 과정을 단계별로 살펴보자.

[단계 1]에서 [단계 5]까지는 행과 열에서 서로 일치하는 부분문자열의 편집 거리를 구하는 과정으로 최장 공통 부분 문자열을 계산하는 알고리즘과 동일하며, [단계 6]부터 [단계 12]까지는 최대 반복자 후보들 중에서 중복되는 문자열을 제외시키는 과정이다.

예를 들어, (그림 7)의 문자열에 대해 [단계 1]에서 [단계 5]까지가 수행되면, (5, 1, 3) (1, 5, 3) (2, 8, 2) (8, 2, 2) (6, 8, 2) (8, 6, 2)와 같이 6개의 *(i, j, Distance)* 값이 생성된다.

[단계 6]은 행 번호와 편집 거리가 같은 후보들을 연속되는 프로세서에 저장하기 위해 <*i, Distance, j*> 크기의 오름차순으로 정렬한다. [18]의 정렬 알고리즘을 이용할 때 $O(1)$

시간 걸리며, 정렬된 결과는 계층 0에 속하는 프로세서에 row-major 순서로 하나씩 저장된다. 즉, $PE(0,0,0)$, $PE(0,0,1)$, $\cdots$, $PE(0,1,0)$, $PE(0,1,1)$, $\cdots$, $PE(0,n-1,n-1)$의 순서로 저장된다.

[단계 7]은 [14]의 기본적인 연산으로 $n\times n\times n$ RMESH를 $n\times n^2$ RMESH의 개념을 가진 구조가 되도록 재구성한다. 이 연산은 계층 0에 row-major 순서로 저장된 <*i, Distance, j*> 값들을 일련의 연속된 값들로 재구성하기 위해 사용되며, 계층 0의 행 i에 있는 위치 코드들을 계층 i의 행 0으로 이동시킨 후, 홀수 번째 계층에 있는 위치 코드를 역순으로 permute함으로써 만들어진다.이 과정을 통하여 각 계층의 프로세서들은 인접한 계층의 프로세서들과 맨 좌측, 혹은 맨 우측에서 연결된 형태로 재구성된다. 이러한 구성은 n^2 개의 프로세서들이 n 번 연속적으로 연결된 형태이며, 이 단계는 $O(1)$ 시간에 수행 가능하다[14].

[단계 8]에서 각 프로세서는 자신의 <*i, Distance*> 값과 왼쪽 프로세서의 <*i, Distance*> 값을 비교하여 다르면 *Live = 1* 을 추가하고, 같으면 *Live = 0* 을 추가하여 *(i, j, Distance, Live)* 를 저장한다. 그리고 맨 앞 프로세서는 무조건 *Live = 1* 로 저장한다. 여기서 *Live = 1* 인 프로세서는 하나의 세그먼트의 시작을 의미하며, 행 번호와 편집거리가 같은 후보들 중 하나만을 남기고 나머지들을 제외시키는 작업에 해당한다. 이 단계에서는 프로세서들이 왼쪽 프로세서만을 접근하여 계산을 하기 때문에 소요 시간은 $O(1)$ 이다.

예를 들어, [단계 5]에서 생성된 6개의 값에 대해 [단계 6]에서 [단계 8]까지의 과정이 수행되면, (그림 8)과 같게 된다.

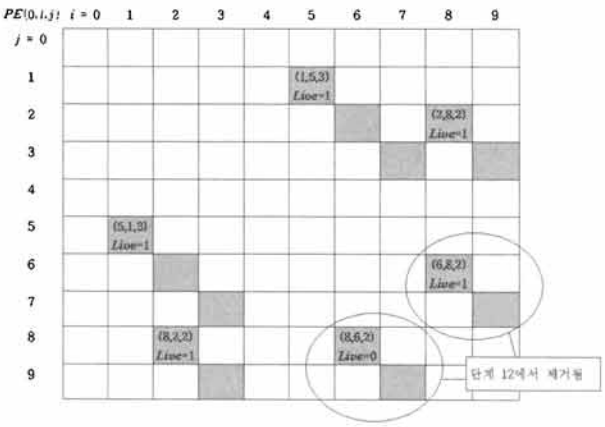
[단계 9]에서는 *(i, j, Distance, Live)* 값을 계층 0의 프로세서에 row-major 순서로 저장하는 단계로서, 먼저 계층 i의 행 0에 있는 위치코드들을 계층 0의 행 i로 이동시킨다. 이 과정은 단계 7의 과정을 역순으로 수행할 수 있으며, 소요 시간은 $O(1)$이다.

[단계 10]에서 *(i, j, Distance, Live)* 값은 계층 0의 프로세서 $PE(0,i,j)$로 이동한다. 여기서 이 값을 받게 되는 $PE(0,i,j)$는 편집거리가 2 이상인 세그먼트의 첫 번째 프로세서에 해당되며, *Live = 0* 인 경우에는 제외되는 값을 의미한다. 이 단계는 [14]의 기본적인 연산에 해당하며 $O(1)$ 시간에 수행된다.

예를 들어, (그림 8)의 값들이 단계 10이 수행된 후에 (그림 9)와 같이 이동된다.

(i, j, Distance) :	(1, 5, 3)	(2, 8, 2)	(5, 1, 3)	(6, 8, 2)	(8, 2, 2)	(8, 6, 2)
<*i, Distance, j*> :	(1, 3, 5)	(2, 2, 8)	(5, 3, 1)	(6, 2, 8)	(8, 2, 2)	(8, 2, 6)
Live :	1	1	1	1	1	0

(그림 8) 단계 8까지 수행된 예



(그림 9) (*i, j, Distance*)와 Live 값이 계층 0의 프로세서 $PE(0,i,j)$로 이동한 후의 예

[단계 11]에서는 프로세서 $PE(0,i,j)$가 자신의 *(i, j, Distance, Live)* 값을 계층 0에서 대각선에 대해 대칭인 프로세서 $PE(0,j,i)$로 자신을 값을 전달하는 과정이다. 이 과정은 *(i, j, Distance, Live)* 값에서 i 와 j 의 값을 상호 교환한 후에 단계 10과 같은 과정을 수행한다. 즉 *(j, i, Distance, Live)* 값을 계층 0의 프로세서 $PE(0,j,i)$로 이동하는 것으로 단계 10과 같이 $O(1)$ 시간에 수행된다.

[단계 12]에서는 계층 0의 프로세서 $PE(0,i,j)$가 자신의 *(i, j, Distance, Live)* 값과 단계 11에서 전달받은 값을 비교하여 두 개의 *Live* 값이 모두 1 인 경우에 *(i, j, Distance)* 값을 저장하고, 그렇지 않으면 이 값을 제거한다. 이 단계에서는 두 값을 비교만 하므로 $O(1)$ 시간에 수행된다.

예를 들어, (그림 9)에서 (8, 6, 2)의 *Live* 값이 0 이므로 (8, 6, 2)와 (6, 8, 2)는 제거되고, 마지막까지 남은 4개의 값 (1, 5, 3) (2, 8, 2) (5, 1, 3) (8, 2, 2) 이 최대 반복자가 된다. 그러나 이들 값은 대각선을 경계로 대칭이므로 대각선 아래쪽의 값을 취하면 (1, 5, 3) (2, 8, 2)이 되고, 각각 부분 문자열 "abc"와 "bc"를 나타낸다.

지금까지 알고리즘 2의 각 단계가 모두 $O(1)$ 시간에 수행될 수 있음을 설명하였으며, 정리 2와 같이 요약할 수 있다.

[정리 2] 주어진 문자열의 길이가 n 일 때, $n\times n\times n$ 3차원 RMESH 구조에서 최대 반복자를 구하는 알고리즘은 $O(1)$ 시간에 수행된다.

5. 결 론

최장 공통 부분문자열 문제는 두 개 이상의 문자열에서 가장 길게 일치하는 부분문자열을 찾는 연산이고 최대 반복자 문제는 하나의 문자열에서 두 번 이상 반복되는 부분문

자열을 찾는 것이다. 이러한 문자열 연산들은 패턴 매칭, 유사도 측정 등의 문자열 처리 분야에서 중요하게 사용되고 있다.

본 논문에서는 3-차원 $n \times n \times n$ RMESH 구조에서 두 문자열의 최장 공통 부분문자열을 계산하는 상수 시간 알고리즘과 최대 반복자를 상수 시간에 찾는 알고리즘을 제안하였다. 이 알고리즘들은 3-차원 RMESH 구조에서 구조적인 특성을 사용함으로써 $O(1)$ 상수 시간 복잡도를 가진다.

참 고 문 헌

[1] R. A. Wagner, Michael J. Fischer, "The String-to-String Correction Problem," Journal of the ACM (JACM), Vol.21, Issue 1, pp.168-173, 1974.

[2] Dan. Gusfield, Algorithms on Strings, Trees, and Sequences, Computer Science and Computational Biology, CamBridge University Press, 1997.

[3] S. Needleman, C. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," J. Mol. BioInformatics, 48(3), 443-453, 1970.

[4] W. Masek and M. Paterson, "A fast algorithm computing string edit distances," J. of Computer and System Sciences, 20(1), pp.13-31, 1980.

[5] 전정은, 박희진, 김동규, "써픽스 배열을 이용한 최장 공통 부분 스트링 계산," 한국정보과학회 가을 학술발표논문집, Vol.4, No.2, pp.739-741, 2004.

[6] I. Lee, C. Illiopoulos, K. Park, "Linear time algorithm for the longest common repeat problem," J. of Discrete Algorithms, 5, pp.243-249, 2006.

[7] A. Apostolico, C. Iliopoulos, G. Landau, BSchieber, and U. Vishkin, "Parallel Construction of a Suffix Tree with Application," Algorithmica, Vol.3, pp.347-365, 1988.

[8] G. Landau, U. Vishkin, "Fast parallel and serial approximate string matching," J. of Algorithms, Vol.10, pp.157-169, 1989.

[9] Y. Jiang, A. Wright, "O(k) parallel algorithms for approximate string matching," Neurla, Parallel and Scientific Computations, Vol.1, pp.443-452, 1993.

[10] Z. Galil, "A Constant Time Optimal Parallel String Matching Algorithm," J. of ACM, Vol.42, pp.908-918, 1995.

[11] H. Lee, F. Ercal, "RMESH Algorithms for Parallel String Matching," International Symposium on Parallel Architectures, Algorithms and Networks, pp.223-226, 1997.

[12] A. Datta, S. Subbiah, "Constant Time Algorithms for String Processing on the Reconfigurable Mesh," The 4th Australasian conference on Parallel and real-time systems, pp.226-237, 1997.

[13] R. Miller, V. Prasanna-Kumar, D. Reisis, and Q. Stout, "Parallel Computation on Reconfigurable Meshes," IEEE Transactions on Computers, Vol.42, No.6, pp.678-692, 1993.

[14] 김기원, 우진운, "선형 사진트리로 표현된 이진 영상의 면적과 둘레 길이를 계산하기 위한 상수시간 RMESH," 한국정보처리학회 논문지, 제5권, 제7호, pp.1746-1758, 1998.

[15] 김경훈, 우진운, "RMESH 구조에서 unaligned 선형사진트리의 alignment를 위한 상수시간 알고리즘," 정보과학회논문지, 제31권 1,2호, pp.10-18, 2004.

[16] J. Jang, H. Park, and V. Prasanna, "A Fast Algorithm for Computing Histogram on a Reconfigurable Mesh," IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol.17, No.2, pp.97-106, 1995.

[17] Y. Ben-Asher, D. Peleg, R. Ramaswami, and A. Schuster, "The Power of Reconfiguration," Journal of Parallel and Distributed Computing, 13, pp.139-153, 1991.

[18] M. Nigam and S. Sahni, "Sorting n Numbers On $n \times n$ Reconfigurable Meshes With Buses," Proceedings 7th International Parallel Processing Symposium, pp.174-181, 1993.

한 선 미

e-mail : hseonmi@dankook.ac.kr

1998년 한국방송통신대학교 경영학과(학사)

2002년 단국대학교 컴퓨터과학전공(석사)

2006년 3월~현 재 단국대학교 컴퓨터과학 전공 박사과정

관심분야 : 병렬알고리즘, 분산 및 병렬처리, 컴퓨터 이론

우 진 운

e-mail : jwwoo@dankook.ac.kr

1980년 서울대학교 수학교육과(학사)

1989년 미국 University of Minnesota 전산학과(박사)

1980년~1983년 대한항공 및 국토개발연구원 전산실 근무

1989년~현 재 단국대학교 정보컴퓨터학부 교수

관심분야 : 병렬알고리즘, 분산 및 병렬처리, 인터넷 응용