DOI: 10.3745/KIPSTA.2010.17A.5.221

접미사 배열을 이용한 Suffix-Prefix가 일치하는 모든 쌍 찾기

한 선 미[†] · 우 진 운[††]

요 약

최근 문자열 연산들이 계산 생물학 및 인터넷의 보안, 검색 분야에 응용되면서 효율적인 문자열 연산을 위한 다양한 자료구조와 알고리즘이 연구되고 있다. suffix-prefix가 일치하는 모든 쌍 찾기는 두 개 이상의 문자열이 주어질 때 각 쌍의 문자열에 대해 가장 긴 suffix와 일치하는 prefix를 찾는 것으로 가장 짧은 슈퍼스트링을 검출하는 근사 알고리즘에서 사용될 뿐만 아니라 생물정보학, 데이터 압축 분야에서도 중요하게 사용된다.

본 논문에서는 접미사 배열을 이용하는 suffix-prefix가 일치하는 모든 쌍 찾기 알고리즘을 제안하며 $O(k \cdot m)$ 시간 복잡도를 가진다. 접미사 배열 알고리즘이 접미사 트리 알고리즘 보다 소요 시간과 메모리 면에서 더 우수함을 실험을 통해서 제시한다.

키워드 : Suffix-Prefix 매칭, 문자열 매칭, 접미사 배열, 접미사 트리

Finding All-Pairs Suffix-Prefix Matching Using Suffix Array

Seon Mi Han[†] · Jin Woon Woo[††]

ABSTRACT

Since string operations were applied to computational biology, security and search for Internet, various data structures and algorithms for computing efficient string operations have been studied. The all-pairs suffix-prefix matching is to find the longest suffix and prefix among given strings. The matching algorithm is importantly used for fast approximation algorithm to find the shortest superstring, as well as for bio-informatics and data compressions.

In this paper, we propose an algorithm to find all-pairs suffix-prefix matching using the suffix array, which takes $O(k \cdot m)$ time complexity. The suffix array algorithm is proven to be better than the suffix tree algorithm by showing it takes less time and memory through experiments.

Keywords : Suffix-Prefix Matching, String Matching, Suffix Array, Suffix Tree

1. 서 론

최근 문자열 연산들이 계산 생물학 분야 및 인터넷의 보안, 검색 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, 멀티 텍스트와 멀티 패턴 문자열 매칭(multi-text/multi-pattern string matching), suffix-prefix가 일치하는 모든 쌍 찾기(all pair suffix-prefix matching) 등이 있다.

suffix-prefix가 일치하는 모든 쌍 찾기는 두 개 이상의 문자열이 주어질 때 각 쌍의 문자열에 대해 가장 긴 suffix와 일치하는 prefix를 찾는 것을 의미한다[1]. suffix-prefix가 일치하는 문자열을 찾는 문제는 하나 또는 여러 개의 문자열에서 존재하는 단일 패턴 또는 멀티 패턴의 모든 위치를 찾고, 존재하는 패턴들의 출현 횟수를 검출하며[2], 가장 짧은 슈퍼스트링(shortest superstring)을 검출하는 근사 알고리즘에서 사용될 뿐만 아니라, [3]에서 연구된 C.elegans의 유전자에서 ACRs의 개수와 위치를 산출하는데 사용되는 등 생물정보학 분야에서도 아주 중요하게 사용된다[1].

이러한 suffix-prefix가 일치하는 모든 쌍 찾기를 위해 일반적인 스트링 매칭이나 접미사 트리(suffix tree)를 이용할 수 있다. 접미사 트리를 이용하는 알고리즘이 시간복잡도

※ 이 연구는 2009학년도 단국대학교 미디어콘텐츠연구원(IMC)의 연구비 지원으로 이루어진 것임
† 정 회 원 : 단국대학교 컴퓨터과학전공 박사과정
†† 종신회원 : 단국대학교 정보컴퓨터학부 교수(교신저자)
논문접수 : 2010년 4월 13일
수 정 일 : 1차 2010년 8월 23일
심사완료 : 2010년 9월 13일

면에서 최적으로 알려져 있다. k 개의 문자열이 주어지고 이 문자열들의 총 길이가 m 일 때, 접미사 트리 알고리즘은 $O(m+k^2)$의 시간복잡도를 가진다[1].

접미사 트리는 문자열의 모든 suffix를 압축된 트라이 형태로 나타낸 것으로, 특정 노드의 모든 하위 노드가 공통의 접두사를 가진다[4, 5]. 또한 모든 내부 노드에 suffix link가 정의되어 있어, 쉽게 부분 문자열을 찾을 수 있다. 그러나 이러한 장점에도 불구하고, 접미사 트리는 연결된 각 노드의 정보와 suffix link 등을 저장하기 때문에 원래의 문자열보다 최소 5배에서 10배 정도의 큰 공간을 요구하므로 DNA 서열 등 대용량의 자료에는 효율적이지 못하다[1].

이러한 단점을 개선하기 위해 접미사 배열(suffix array)이 제안되었다. 접미사 배열은 문자열의 모든 suffix를 사전적 순서로 정렬하여 그 시작 위치를 배열로 나타낸 것으로 배열을 이용하므로 공간을 적게 차지한다[6-9]. 접미사 트리를 이용하여 기존의 애플리케이션들을 대체하기 위한 연구 또한 활발하게 이루어지고 있다[10-14].

또한 특정한 문자열을 탐색할 때 Backward Search를 이용하여 선형 시간만큼 소요하는 탐색 알고리즘도 제안되었다[15-17].

본 논문에서는 접미사 트리에 비해 적은 공간과 사전 처리(preprocessing)의 과정을 거치지 않는 접미사 배열을 이용한 suffix-prefix가 일치하는 모든 쌍 찾기 알고리즘을 제안하고자 한다. 현재 접미사 배열을 이용한 알고리즘은 알려져 있지 않으며, 제안하는 알고리즘은 [17]의 선형 탐색 알고리즘을 활용한다.

본 논문의 구성은 다음과 같다. 2장에서는 접미사 배열에 대해 설명하고 3장에서 suffix-prefix가 일치하는 모든 쌍 찾기를 위한 메모리 구조와 알고리즘에 대해 설명하고 4장에서는 접미사 트리와 비교한 실험 결과를 보인 후 5장에서 결론을 맺는다.

2. 관련 연구

2.1 접미사 배열

접미사 배열은 접미사 트리에 비해 저장 공간을 적게 소모하는 자료 구조로 Manber와 Myers[6]에 의해 제안된 후, 선형시간 내에 생성될 수 있는 여러 가지 알고리즘들이 개발되어 왔으며[7-9], 문자열의 검색 및 매칭, 데이터의 압축, 클러스터링 등 문자열 관련 여러 분야에서 접미사 트리를 대신하고 있다[12].

접미사 배열은 문자열의 모든 suffix들을 사전적 순서(lexicographical order)에 따라 정렬시킨 후 해당 suffix들의 순서를 배열 형태로 저장한다.

길이가 n 인 문자열 $T = T_1\ T_2\ T_3 \ldots T_n$ 는 제한된 수의 알파벳 ($\Sigma = \{\sigma_1, \sigma_2, \sigma_3 \ldots \sigma_{|\Sigma|}\}$) 으로 구성되며, i 번째 위치에서 시작하는 T 의 부분문자열 $T_i = T_i\ T_{i+1}\ T_{i+2} \ldots T_n$ 를 T의 i 번째 suffix라 한다. 또한, 문자열 T의 끝을 표시하기 위해 Σ 의 요소이지만 문자열 T 에는 없는 특별한 기호 '$'를 사용한다.

i	$SA_{[i]}$	$S_{SA[i]}$
1	14	$
2	13	a$
3	12	aa$
4	7	aatataa$
5	1	aattataatataa$
6	10	ataa$
7	5	ataatataa$
8	8	atataa$
9	2	attataatataa$
10	11	taa$
11	6	taatataa$
12	9	tataa$
13	4	tataatataa$
14	3	ttataatataa$

(그림 1) 문자열 T=aattataatataa$ 의 접미사 배열

접미사 배열은 문자열 T 의 모든 suffix들을 사전적 순서에 의해 오름차순으로 정렬한 후 그 위치 값들을 배열 $SA[1..n]$에 저장한 것이다. 예를 들어, 문자열이 T=aattataatataa$ 일 때 접미사 배열은 (그림 1)과 같이 생성된다. (그림 1)에서 두 번째 열의 $SA[i]$ 는 접미사 배열이고, 세 번째 열의 $S_{SA[i]}$는 접미사 배열이 나타내는 문자열의 suffix이다.

접미사 트리는 생성 시간과 검색 및 매칭에 대해 접미사 배열에 비해 빠른 처리 시간을 갖지만 자료 구조가 복잡하며, 원래의 문자열에 비해 큰 크기의 저장 공간을 필요로 하는 단점이 있다. 반면, 접미사 배열은 저장 공간을 적게 사용하고 구조가 간단하여 구현이 쉬운 실용적인 모델이라는 장점이 있으나 생성 시간이 많이 걸리는 단점이 있다.

2.2 Backward Search

문자열 T 에서 패턴 P 를 찾는 스트링 매칭 문제를 해결하기 위해 Backward Search 방법을 사용할 수 있다[15].

알파벳 $\Sigma = \{\sigma_1, \sigma_2, \sigma_3 \ldots \sigma_{|\Sigma|}\}$ 상에서 정의된 길이 n 인 문자열 T 와 길이 m 인 패턴 P 가 주어져 있다고 하자. 문자열 T 의 모든 suffix들을 사전적 순서로 정렬해 놓은 접미사 배열 SA 에서 패턴 P 를 prefix로 가지는 모든 suffix들을 찾음으로써 패턴 P 가 문자열 T 에서 어느 위치에 몇 번 나오는가를 계산할 수 있다.

접미사 배열은 사전적 순서에 의해 정렬되어 있으므로 해당 패턴이 들어있는 부분들이 블록화되어 있다. 이러한 성질을 이용하여 해당 블록의 시작부분과 마지막부분의 인덱스를 조정하면서 패턴의 존재 유무 및 위치와 개수까지 찾을 수 있다. 알고리즘 1은 문자열 T 에서 패턴 P 를 찾는 Backward Search 알고리즘을 보여주고 있으며[17], (그림 2)는 패턴 P=tat이고, 문자열 T=aattataatataa$ 일 때 알고리즘 1에서 배열 C의 값과 접미사 배열 SA의 관계를 보여준다.

배열 $C[1..|\Sigma|]$에는 문자열 T에 유일하게 존재하는 각

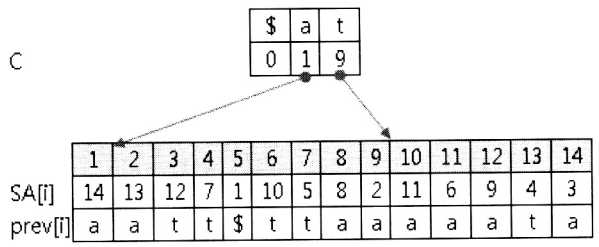
```
Algorithm BW_Count(P[1,p])
1. c = P[p], i=p;
2. sp = C[c]+1, ep = C[c + 1];
3. while ((sp≤ep) and (i≥2) do
4.          c = P[i - 1];
5.          sp = C[c] + Occ(c,1,sp-1) +1;
6.          ep = C[c] + Occ(c,1,ep);
7.          i=i-1;
8. if(ep<sp) then return  "pattern not found"
       else return "found (ep - sp +1) occurrences"
```

(알고리즘 1) Backward Search 알고리즘

문자가 접미사 배열 SA에서 처음으로 나타나는 위치 바로 앞의 인덱스를 저장하며, 이것은 알고리즘 1의 Backward Search에서 sp와 ep를 결정하는데 사용된다. sp는 블록의 시작 인덱스이고, ep는 블록의 끝 인덱스에 해당한다. $Occ(c,i,j)$ 함수는 접미사 배열 $SA[i]$와 $SA[j]$ 사이의 suffix들의 바로 앞 문자(문자열 T에서의 앞 문자로 (그림 2)에서 배열 $prev$ 에 저장됨)들 중 문자 c의 수를 계산한다. Occ 함수의 리턴 값을 이용하여 접미사 배열에서 패턴이 나타나는 블록의 시작 인덱스인 sp 와 끝 인덱스인 ep 를 계산한다.

예를 들어, 문자열 T=aattataatataa$에 대해 패턴 P=tat를 검색해보자. 패턴 P의 마지막 문자 $P[3]$=t부터 검색하기 시작하며, 문자 t 에 해당하는 배열 C의 값은 9이므로 sp=9+1=10, 알파벳 집합 Σ에서 t가 가장 큰 문자이므로 ep는 T의 길이인 14가 된다. $P[2]$=a에 대해 while 반복문 내에서 sp를 결정하기 위해 $Occ('a',1,10-1)$ 함수를 호출하면, 배열 $prev[1]$과 $prev[9]$ 사이에 'a'가 4개 있으므로 sp=1+4+1=6 이 되고, ep를 결정하기 위해 $Occ('a',1,14)$ 함수를 호출하면, 배열 $prev[1]$과 $prev[14]$ 사이에 'a'가 8개 있으므로 ep=1+8=9 가 된다. 다음으로, 마지막 while 반복문 내에서 $P[1]$=t 가 되며 sp를 결정하기 위해 $Occ('t',1,6-1)$ 함수를 호출하면, 배열 $prev[1]$과 $prev[5]$ 사이에 't'가 2개 있으므로 sp=9+2+1=12 가 되고, ep를 결정하기 위해 $Occ('t',1,9)$ 함수를 호출하면, 배열 $prev[1]$과 $prev[9]$ 사이에 't'가 4개 있으므로 ep=9+4=13 이 된다.

이것은 패턴 P=tat 가 $SA[12]$와 $SA[13]$ 사이의 suffix의 prefix가 됨을 의미하며, 2번 나타나고, 그 위치는 문자열 T에서 $SA[12]$ = 9, $SA[13]$ = 4 에 존재함을 의미한다. 즉 문자열 aattataatataa$ 와 aattataatataa$ 에서 밑줄 친 부분



(그림 2) 배열 C의 값과 접미사 배열 SA의 관계

문자열이 일치하는 패턴에 해당한다.

알고리즘 1은 $O(m \cdot |\Sigma|)$ 비트 공간을 사용하여 패턴의 길이에 비례하는 $O(m)$ 시간에 스트링 매칭을 수행한다[17]. Suffix array를 구성하기 위해 선형시간 알고리즘[7-9]을 이용할 때 $O(m)$ 시간과 $O(m)$ 공간이 필요하다.

3. Suffix-Prefix가 일치하는 모든 쌍 찾기

3.1 정의 및 구조

suffix-prefix가 일치하는 모든 쌍 찾기는 두 개 이상의 문자열이 주어질 때 각 쌍의 문자열에 대해 가장 긴 suffix와 일치하는 prefix를 찾는 문제이다[1].

예를 들어, 세 개의 문자열 S_1=xbaxab, S_2=abxb, S_3=axabaxba에 대해 suffix-prefix가 일치하는 모든 쌍 찾기의 결과는 다음과 같다. S_1과 S_2의 suffix-prefix는 'ab', S_1 과 S_3의 suffix-prefix는 'axab'가 되며, S_2 와 S_1의 suffix-prefix는 'xb', S_2 와 S_3의 suffix-prefix는 존재하지 않는다. 마지막으로, S_3 와 S_1의 suffix-prefix는 'xba'가 되고, S_3 와 S_2의 suffix-prefix는 'a'가 된다. 이러한 결과를 표시하면 <표 1>과 같게 된다. <표 1>에서 숫자는 S_i 와 S_j 의 suffix-prefix가 일치하는 S_i의 가장 긴 suffix의 시작 위치에 해당한다. 그리고 0은 일치하는 suffix-prefix가 존재하지 않음을 의미한다.

suffix-prefix가 일치하는 모든 쌍 찾기를 위해 k 개의 문자열, S_1, S_2, ..., S_k 가 주어질 때, 각 문자열의 길이를 n_i $(1 \le i \le k)$ 라 하자. 접미사 배열을 이용하기 위해 k 개의 문자열을 연결하여 하나의 긴 문자열 $S = \{ S_1, S_2, ..., S_k \}$ 을 생성하며, 문자열들을 구별하기 위해 연결되는 문자열 사이에 특수 문자 '$'을 넣는다.

suffix-prefix가 일치하는 모든 쌍 찾기를 위해 본 논문에서 제안하는 알고리즘은 2.2절의 Backward Search 알고리즘을 변형하여 사용하고, 다섯 개의 배열을 사용한다. 먼저 문자열 S 의 길이를 m ($m = \sum_{i=1}^{k}(n_i+1)$) 이라 하자. SA는 S의 접미사 배열을 저장하며, 배열 $prev$는 직전 문자를 저장한다. 배열 $base$는 각 문자가 접미사 배열 SA에서 처음으로 나타나는 위치의 인덱스를 저장하는데, 2.2절의 Backward Search 알고리즘에서 사용된 배열 C 에 해당한다. 배열 $prePos$는 S 내에서 문자열 S_1, S_2, ..., S_k 의 시작위치를 저장하고, 배열 SA_prePos는 $prePos$ 배열의 값(즉, S 내에서 문자열 S_1, S_2, ..., S_k 의 시작위치)이 배열 SA에서 나타나는 위치의 인덱스를 저장한다. 이때 배열 SA 와 $prev$ 의 크기

〈표 1〉 suffix-prefix가 일치하는 모든 쌍 찾기의 예

	S_1=xbaxab	S_2=abxb	S_3=axabaxba
S_1=xbaxab	-	5	3
S_2=abxb	3	-	0
S_3=axabaxba	6	8	-

	$	a	b	x
base	0	3	10	16

	$1	$2	$3	전체문자열
	xbaxab$	abxb$	axabaxba$	xbaxab$abxb$axabaxba$
prePos	1	8	13	
SA_prePos	21	7	9	

	i	SA	prev[i]	$SA_{[i]}$
	1	21	a	$
	2	7	b	$abxb$axabaxba$
	3	12	b	$axabaxba$
	4	20	b	a$
	5	5	x	ab$abxb$axabaxba$
	6	15	x	abaxba$
S_2	7	8	$	abxb$axabaxba$
	8	3	b	axab$abxb$axabaxba$
S_3	9	13	$	axabaxba$
	10	17	b	axba$
	11	6	a	b$abxb$axabaxba$
	12	11	x	b$axabaxba$
	13	19	x	ba$
	14	2	x	baxab$abxb$axabaxba$
	15	16	a	baxba$
	16	9	a	bxb$axabaxba$
	17	4	a	xab$abxb$axabaxba$
	18	14	a	xabaxba$
	19	10	b	xb$axabaxba$
	20	18	a	xba$
S_1	21	1	a	xbaxab$abxb$axabaxba$

(그림 3) S={S1=xbaxab, S2=abxb, S3=axabaxba}일 때의 예

는 m 이며, 배열 *base*의 크기는 사용되는 알파벳의 수, $|\Sigma|$에 해당하며, *prePos, SA_prePos* 배열의 크기는 문자열의 수인 k 에 해당한다.

예를 들어, <표 1>의 S_1=xbaxab, S_2=abxb, S_3=axabaxba에 대해 배열 *SA, prev, base, prePos, SA_prePos* 를 생성하면 (그림 3)과 같게 된다. 이때 S=xbaxab$abxb$axabaxba$가 된다.

3.2 알고리즘

k 개의 문자열, S_1, S_2, ..., S_k 을 연결하여 하나의 문자열 S를 만들고 접미사 배열 SA를 생성한 후, 3.1절에서 설명된 나머지 배열들의 값을 결정한다.

이 배열들을 매개변수로 받아 suffix-prefix가 일치하는 모든 쌍 찾기를 수행하며, 알고리즘을 요약하면 알고리즘 2와 같다. suffix-prefix가 일치하는 모든 쌍 찾기의 결과는 배열 *SufPref* 에 저장되어 반환된다. 배열 *SufPref* 는 크기가 $k \times k$ 로서 <표 1>과 같이 결과를 저장하며, *SufPref[i][j]* 는 S_i 와 S_j 의 suffix-prefix가 일치하는 S_i의 가장 긴 suffix의 시작 위치에 해당한다.

알고리즘 2의 4행에서 16행 사이의 for 반복문은 각 문자열에 대해 suffix-prefix가 일치하는 모든 쌍을 찾기 위해 실행된다. for 반복문내의 5행에서 15행은 알고리즘 1의 Backward Search 알고리즘을 약간 변형한 것으로, 8행에서 10행의 for 반복문이 추가되었다. 이 반복문은 Backward

```
function SuffixPrefix(SA[], base[], prev[], SA_prePos[][], k)
{
 1. global S1, S2, ... , Sk ;     // 각 문자열은 전역변수
 2. int SufPref[k][k] ;           // k : 문자열의 수
 3. initialize SufPref[][] with 0's ; // 0으로 초기화
 4. for(i=1; i <= k; i++){  // 각각의 문자열 Si 에 대해 반복
 5.    n = length(Si) ;   c = Si[n];
 6.    sp = base[c]+1;  ep = base[c+1];
 7.    while ((sp <= ep) and (n >= 2)) {
 8.       for(j=1; j <= k; j++) // 각 문자열에 대해 시작 위치를 확인
 9.             if (SA_prePos[j]>=sp && SA_prePos[j]<=ep
10.                SufPref[i][j] = n ;
11.          n=n-1 ;
12.       c = Si[n];
13.       sp = base[c]+Occ(c, 1, sp-1)+1;
14.       ep = base[c]+Occ(c, 1, ep);
15.    } // end of while
16.  } // end of for
17.  return SufPref ;
} // end of function
```

(알고리즘 2) suffix-prefix가 일치하는 모든 쌍 찾기 알고리즘

Search의 실행 과정에서 블록의 시작 인덱스 *sp*와 끝 인덱스 *ep* 사이에 문자열의 시작 위치가 존재하는가를 확인하여 prefix를 갖는 문자열을 찾기 위한 것이다. 즉, *sp*와 *ep*가 나타내는 범위 내에 *SA_prePos* 배열의 값이 존재한다면 현재의 검색 대상 문자의 위치를 인덱스로 하는 suffix와 prefix가 일치하는 문자열이 존재하는 것을 의미한다.

예를 들어, (그림 3)의 S={S_1=xbaxab, S_2=abxb, S_3=axabaxba}에 대해 그 과정을 알아보자. 4행의 for 반복문은 세 개의 문자열에 대해 3번 반복 수행된다.

k = 1 일 때, 문자열 S_1의 길이를 계산하여 n에 할당하면 n은 6이 되고 S_1*[6]*의 문자는 'b' 이므로 Backward Search에서 'b'를 prefix로 하는 블록의 시작 위치 *sp* = *base*['b']+1 = 11 과 끝 위치 *ep* = *base*['x'] = 16으로 결정된다. 알고리즘 2의 8행과 10행 사이에서 11부터 16 사이에 *SA_prePos* 배열의 값이 존재하는가를 검사하는데 *SA_prePos[1]*= 21, *SA_prePos[2]*= 7, *SA_prePos[3]*= 9 이므로 이 범위 내에 있는 값은 없다.

다음으로, n을 1만큼 감소시키면 n은 5가 되며 S_1*[5]*= 'a'가 되어 *Occ* 함수를 호출하여 새로운 *sp*와 *ep*를 계산한다. *Occ('a',1,11-1)* 을 호출하면 'a'가 *prev* 배열의 1번째에서 10번째 위치까지 1번 나타나므로 1이 반환되어 *sp=base*['a']+1+1= 5, *Occ('a',1,16)* 을 호출하면 'a'가 *prev* 배열의 1번째에서 16번째 위치까지 4번 나타나므로 4가 반환되어 *ep=base*['a']+4=7 로 계산된다. 이것은 알고리즘 2의 7행의 while 반복문의 다음 반복을 위한 새로운 블록의 시작과 끝 인덱스가 된다.

계속해서 5와 7 사이에서 *SA_prePos*의 값이 존재하는가를 검사하는데 *SA_prePos[2]*= 7 만이 이 범위 내에 있다. 이것은 S_1의 suffix 'ab'와 일치하는 prefix를 가진 문자열이 S_2임을 의미하며 *SufPref[1][2]=5* 로 결정된다. 이 과정을 그림으로 나타내면 (그림 4)와 같다.

다음으로, n을 1만큼 감소시키면 n은 4가 되며 S_1*[4]*= 'x'

	i	SA	prev[i]	$SSA_{[i]}$
	1	21	a	$
	2	7	b	$abxb$axabaxba$
	3	12	b	$axabaxba$
	4	20	b	a$
sp_2 →	5	5	x	ab$abxb$axabaxba$
	6	15	x	abaxba$
ep_2 →	7	8	$	abxb$axabaxba$
	8	3	b	axab$abxb$axabaxba$
	9	13	$	axabaxba$
	10	17	b	axba$
sp_1 →	11	6	a	b$abxb$axabaxba$
	12	11	x	b$axabaxba$
	13	19	x	ba$
	14	2	x	baxab$abxb$axabaxba$
	15	16	a	baxba$
ep_1 →	16	9	a	bxb$axabaxba$
	17	4	a	xab$abxb$axabaxba$
	18	14	a	xabaxba$
	19	10	b	xb$axabaxba$
	20	18	a	xba$
	21	1	a	xbaxab$abxb$axabaxba$

(그림 4) S_1의 suffix 'ab'와 S_2의 prefix 'ab'를 찾는 과정

가 되어 *Occ('x',1,5-1)* 을 호출하면 'x'가 *prev* 배열의 1번째에서 4번째 위치까지 존재하지 않으므로 0이 반환되어 *sp=base*['x']+0+1= 17, *Occ('x',1,7)* 을 호출하면 'x'가 *prev* 배열의 1번째에서 7번째 위치까지 2번 나타나므로 2가 반환되어 *ep*=base['x']+2= 18 로 계산된다. 이것은 인덱스 17과 18이 새로운 블록의 범위가 됨을 의미한다.

계속해서 17과 18 사이에서 *SA_prePos*의 값이 존재하는가를 검사하는데 이 범위에 속하는 *SA_prePos* 값이 없다.

다음으로, *n*을 1만큼 감소시키면 *n*은 3이 되며 $S_1[3]$= 'a'가 되어 *Occ* 함수를 호출하여 새로운 *sp*와 *ep*를 계산한다. *Occ('a',1,17-1)* 을 호출하면 'a'가 *prev* 배열의 1번째에서 16번째 위치까지 4번 나타나므로 4가 반환되어 *sp=base*['a']+4+1= 8, *Occ('a',1,18)* 을 호출하면 'a'가 *prev* 배열의 1번째에서 18번째 위치까지 6번 나타나므로 6이 반환되어 *ep*=base['a']+6= 9 로 계산된다.

계속해서 8과 9사이에서 *SA_prePos[3]*= 9 가 이 범위 내에 있다. 이것은 S_1의 suffix 'axab'와 일치하는 prefix를 가진 문자열이 S_3임을 의미하며 *SufPref[1][3]=3* 로 결정된다. 이 과정을 그림으로 나타내면 (그림 5)와 같다.

다시 *n*을 1만큼 감소시키면 *n*은 2가 되며 $S_1[2]$= 'b'가 되어 *Occ* 함수를 호출하여 새로운 *sp*와 *ep*를 계산한다. *Occ('x',1,8-1)* 을 호출하면 'b'가 *prev* 배열의 1번째에서 7번째 위치까지 3번 나타나므로 3이 반환되어 *sp=base*['b']+3+1= 14, *Occ('b',1,9)* 을 호출하면 'b'가 *prev* 배열의 1번째에서 9번째 위치까지 4번 나타나므로 4가 반환되어 *ep=base*['b']+4= 14 로 계산된다. 14와 14 사이에서 *SA_prePos*의 값이 존재하는가를 검사하는데 이 범위에 속하는 *SA_prePos* 값이 없다.

	i	SA	prev[i]	$SSA_{[i]}$
	1	21	a	$
	2	7	b	$abxb$axabaxba$
	3	12	b	$axabaxba$
	4	20	b	a$
	5	5	x	ab$abxb$axabaxba$
	6	15	x	abaxba$
	7	8	$	abxb$axabaxba$
sp_3 →	8	3	b	axab$abxb$axabaxba$
ep_3 →	9	13	$	axabaxba$
	10	17	b	axba$
	11	6	a	b$abxb$axabaxba$
	12	11	x	b$axabaxba$
	13	19	x	ba$
	14	2	x	baxab$abxb$axabaxba$
	15	16	a	baxba$
	16	9	a	bxb$axabaxba$
sp_1 →	17	4	a	xab$abxb$axabaxba$
ep_1 →	18	14	a	xabaxba$
	19	10	b	xb$axabaxba$
	20	18	a	xba$
	21	1	a	xbaxab$abxb$axabaxba$

(그림 5) S_1의 suffix 'axab'와 S_2의 prefix 'axab'를 찾는 과정

마지막으로 *n*을 1만큼 감소시키면 *n*은 1이 되며 $S_1[1]$= 'x'가 되어 *Occ* 함수를 호출하여 새로운 *sp*와 *ep*를 계산한다. *Occ('x',1,14-1)* 을 호출하면 'x'가 *prev* 배열의 1번째에서 13번째 위치까지 4번 나타나므로 4가 반환되어 *sp=base*['x']+4+1= 21, *Occ('x',1,14)* 을 호출하면 'x'가 *prev* 배열의 1번째에서 14번째 위치까지 5번 나타나므로 5가 반환되어 *ep=base*['b']+5= 21 로 계산된다. 21과 21 사이에서 *SA_prePos*의 값이 존재하는가를 검사하는데 *SA_prePos[1]*= 21 가 이 범위 내에 있다. 이것은 S_1 자신의 시작위치이므로 무시된다.

지금까지 결정된 값은 *SufPref[1][1]=0, SufPref[1][2]=5, SufPref[1][3]=3* 이 되며, <표 1>의 첫 번째 행과 같게 되다. *SufPref[1][2]=5* 는 S_1=xbaxab 와 S_2=abxb 의 밑줄 친 부분문자열이 최대 suffix 와 최대 prefix가 됨을 나타낸다. 그리고 *SufPref[1][3]=3* 은 S_1=xbaxab 와 S_3=axababxba 의 밑줄 친 부분문자열이 최대 suffix 와 최대 prefix가 됨을 나타낸다.

문자열 S_2, S_3도 위의 과정과 동일한 방법으로 *SufPref* 배열의 두 번째와 세 번째 행에 <표 1>과 같은 값을 저장하게 된다.

Backward Search 알고리즘은 패턴의 길이에 비례하는 $O(m)$ 시간에 스트링 매칭을 수행하므로[17], 알고리즘 2의 7행 while 반복문은 문자열 S_i 의 길이인 n_i만큼 반복하게 되고, 매 반복마다 8행의 for 반복문은 k 번 반복하므로 7행에서 16행의 실행문의 시간복잡도는 $O(k \cdot n_i)$가 된다. 따라서 4행의 for 반복문이 모든 문자열에 대해 반복하므로 알고리즘 2의 시간복잡도는 $O(k \cdot m)$이 된다. 이때 $m = \sum_{i=1}^{k} n_i$ 이며, k 개 문자열의 길이 합이다.

한편, 접미사 트리 알고리즘은 $O(m+k^2)$의 시간복잡도를 가지나, 연결된 각 노드의 정보와 suffix link 등을 저장하기 때문에 많은 공간을 사용하는 단점이 있다[1].

4. 실험 결과

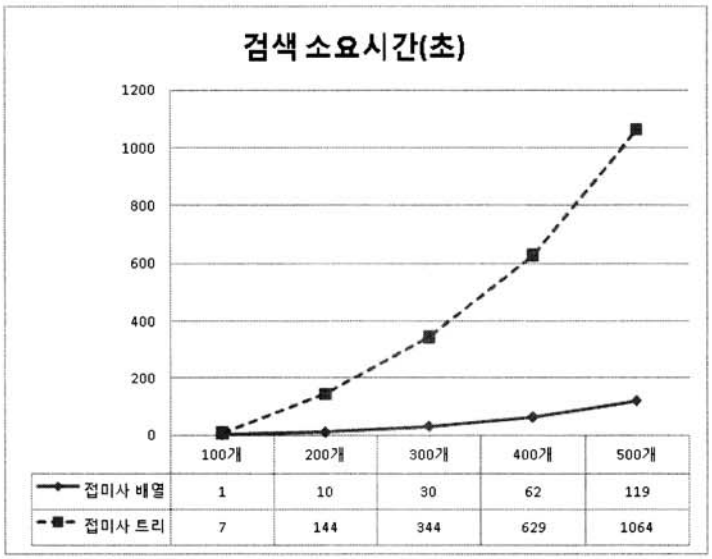
본 절에서는 접미사 배열과 접미사 트리를 사용하여 suffix-prefix가 일치하는 모든 쌍 찾기를 위해 소요되는 시간과 메모리를 비교하였다. 접미사 배열은 Käarkkäainen-Sanders 알고리즘[8]을 사용하여 생성한 후 알고리즘 2를 적용하였으며, 접미사 트리는 Ukkonen 알고리즘[5]을 사용하여 생성한 후에 [1]의 suffix-prefix 매칭을 위한 코드를 사용하였다.

실험 데이터로는 DNA 문자열로 길이가 500에서 1000 사이인 문자열을 100개, 200개, 300개, 400개, 500개로 실험하였으며 하나의 문자열은 알파벳 $\Sigma = \{a, c, g, t, \$\}$로 구성되어 있다. 그리고 모든 실험은 Pentium(R) D CPU 3.40GHz, 1.5GB 메모리, Windows XP 환경에서 C++로 구현하여 사용하였다.

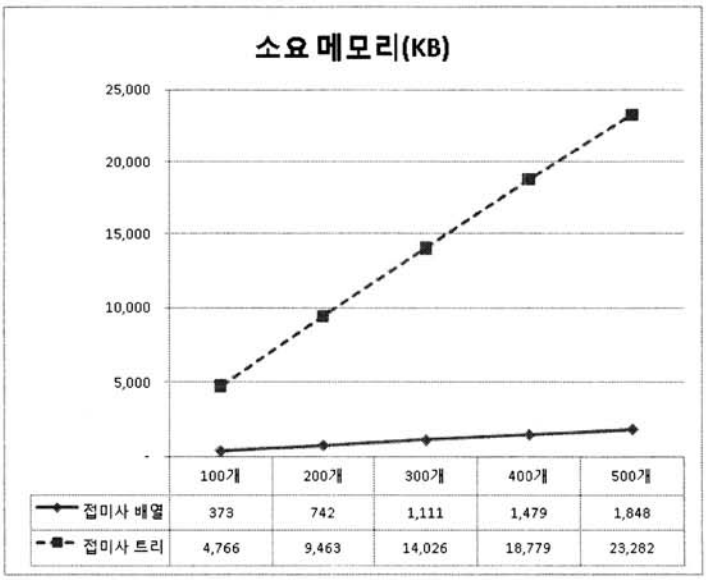
(그림 6)은 실험에 걸린 소요시간을 그래프로 보여준다. 소요시간은 접미사 트리와 접미사 배열을 구성하는데 필요한 시간을 포함하며, 문자열의 수가 많아질수록 접미사 트리 알고리즘이 접미사 배열 알고리즘보다 시간을 더 많이 소요하는 것을 알 수 있다.

(그림 7)은 접미사 배열과 접미사 트리의 소요 메모리를 보여준다.

접미사 배열의 경우는 3.1절에서 설명된 다섯 개의 배열



(그림 6) 접미사 배열과 접미사 트리에서의 소요시간 비교



(그림 7) 접미사 배열과 접미사 트리의 소요 메모리 비교

에 소요되는 메모리를 측정하여 Kilo Byte 단위로 표현하였다. 접미사 트리의 경우, 접미사 트리를 표현하기 위해 필요한 모든 노드의 메모리를 합하여 Kilo Byte 단위로 표현하였다. 하나의 노드는 부모 노드, 첫 번째 자식노드, 왼쪽과 오른쪽 형제 노드를 가리키는 포인터 필드, 그리고 트리 순회에 사용되는 정보들을 포함하는 각종 link 필드들로 구성된다.

5. 결 론

suffix-prefix가 일치하는 모든 쌍 찾기는 두 개 이상의 문자열이 주어질 때 각 쌍의 문자열에 대해 가장 긴 suffix와 일치하는 prefix를 찾는 것으로 가장 짧은 슈퍼스트링을 검출하는 근사 알고리즘에서 사용될 뿐만 아니라 생물정보학, 데이터 압축 분야에서도 중요하게 사용된다.

본 논문에서는 접미사 배열을 이용하는 suffix-prefix가 일치하는 모든 쌍 찾기 알고리즘을 제안하였으며, 이 알고리즘은 문자열 매칭을 위한 Backward Search 알고리즘을 이용하고 $O(k \cdot m)$ 시간복잡도를 갖는다. 그러나 접미사 트리 알고리즘은 $O(m+k^2)$의 시간복잡도를 가지나 각 노드의 정보와 suffix link 등을 저장하기 때문에 많은 메모리를 사용하고 이에 따른 시간 지체가 크다.

본 논문에서는 제안한 접미사 배열 알고리즘과 접미사 트리 알고리즘을 실행하여 소요되는 시간 및 메모리를 비교하였다. 접미사 배열 알고리즘이 접미사 트리 알고리즘 보다 저장 공간 면과 시간 면에서 더 우수함을 볼 수 있었다.

참 고 문 헌

[1] D. Gusfield, "Algorithms on Strings, Trees, and Sequences," Computer Science and Computational Biology, CamBridge University Press, 1997.

[2] Z. M. Kedem. G. M. Landau, K. V. Palem, "Parallel suffix-prefix-Matching Algorithm and Applications," SIAM Journal on Computing, Vol.25, No.5, pp.998-1023, 1996.

[3] P. Green, D. Lipman, D. Hillier, R. Waterston, D. States, J. M. Claverie. "Ancient conserved regions in new gene sequences and the protein databases," Science, Vol.259, pp.1711-1716, 1993.

[4] E. M. McCreight, "A Space-Economical Suffix Tree Construction Algorithm," Journal of ther ACM, Vol.23, pp.262-272, 1976.

[5] E. Ukkonen, "On-line construction of suffix trees," Algorithmica, Vol.14, pp.249-260, 1995.

[6] U. Manber, G. Myers, "Suffix arrays: a new method for on-line string searches," SIAM Journal of Computing 22, pp.935-948, 1993.

[7] P. Ko and S. Aluru, "Space efficient linear time construction of suffix arrays," In Proc. of the 14th Annual Symposium on Combinatorial Pattern Matching, Vol.2676, pp.200-210, 2003.

[8] J. Kärkkäinen, P. Sanders, S. Burkhardt, "Linear work suffix array construction," Journal of the ACM, Vol.53, pp.918-936, 2006.

[9] D. K. Kim, J. S. Sim, H. Park, K. Park, "Constructing suffix arrays in linear time," Journal of Discrete Algorithms, Vol.3, pp.126-142, 2005.

[10] R. Grossi, J. S. Vitter, "Compressed suffix arrays and suffix trees with applications to text indexing and string matching," In Proc. of the 32nd ACM Symposium on Theory of Computing, pp.397-406, 2000.

[11] WK. Hon, K. Sadakane, WK. Sung, "Breaking a time-and-space barrier in constructing full-text indices," In Proc. of the 44th Symposium on Foundations of Computer Science, pp.251-260, 2003.

[12] M. Abouelhoda, E. Ohlebusch, S. Kurtz, "Optimal exact string matching based on suffix arrays," In Proc. of the 9th International Symposium on String Processing and Information Retrieval. Vol.2476, pp.31-43, 2002.

[13] K. Sadakane, "Succinct representation of lcp information and improvement in the compressed suffix arrays," In. proc. of the 13th ACM-SIAM Symposium on Discrete algorithms, pp.225-232, 2002.

[14] M. Abouelhoda, S. Kurtz, E. Ohlebusch, "Replacing Suffix Trees with Enhanced Suffix Arrays," Journal of Discrete Algorithms, Vol.2, pp.53-86, 2004.

[15] P. Ferragina, G. Manzini, "Opportunistic data structures with applications," In Proc. of the 41st IEEE Symposium on Foundations of Computer Science, pp.390-398, 2000.

[16] 심정섭, 김동규, 박희진, 박근수, "접미사 배열을 이용한 선형시간 탐색", 정보과학회 논문지: 시스템 및 이론, 제 32권 제 5호, pp.255-259, 2005.

[17] 최용욱, 심정섭, 박근수, "접미사 배열을 이용한 시간과 공간 효율적인 검색", 정보과학회 논문지: 시스템 및 이론, 제 32권 제 5호, pp.260-267, 2005.

한 선 미

e-mail : hseonmi@dankook.ac.kr

1998년 한국방송통신대학교 경영학과(학사)

2002년 단국대학교 컴퓨터과학전공(석사)

2006년 3월~현 재 단국대학교 컴퓨터과학전공 박사과정

관심분야 : 알고리즘, 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 : 알고리즘, 분산 및 병렬처리, 인터넷 응용