Suffix Tree를 이용한 스트링 검색 성능 평가

Performance Evaluation of Searching String Using Suffix Tree

김선영

부산대학교 컴퓨터공학과

s.y.kim@pusan.ac.kr

ABSTRACT

생물 정보학에서 두 시퀀스 간의 공통 스트링을 찾는 일은 생물의 진화 과정을 추측함에 있어 가장 중요한 과정의 하나로 매우 큰 의미를 지닌다. 이 때문에 서로 다른 스트링 간의 공통 부분을 찾는 방법에 대한 연구가 계속되었으며, 현재는 지역 정렬, 전역 정렬 등의 방법이 공통 스트링을 찾기 위한 방법으로 널리 사용되고 있다. 비교적 최근에 재조명 받고 있는 suffix tree라는 자료 구조는 정확히 일치하는 스트링에 한하여 선형 시간에 탐색을 수행할 수 있기 때문에 아주 효과적인 방법으로 거론되고 있다. 본 보고서에서는 suffix tree의 정의와 동작 원리, 구현 방법과 활용법에 대해 알아보고, suffix tree를 이용하여 40∼60MB 가량의 fasta 포맷의 모기 유전체 자료 및 한글 말뭉치 파일을 사용하여 그 성능을 측정하는 실험을 수행하였다. 실험 결과 suffix tree는 비교 대상 스트링의 길이 n과 무관하게 검색할 스트링의 길이 m에 비례하는 O(m) 시간에 구현됨을 확인할 수 있었다. 다만 29MB 이상의 데이터를 바로 집어넣을 경우 Out of Memory 오류가 발생하였는데, 이는 suffix tree의 공간 복잡도에 기인한 것으로 추정된다. 추후 이 부분에 대하여 자세히 조사할 계획이다.

Keywords Suffix Tree, Searching Common String, Linear Time Complexity

1 서 론

생물정보학에서 두 개의 시퀀스 사이에서 공통된 문자열을 찾아내는 것은 아주 중요한 의미를 가진다. 서로 다른 생물의 유전자에서 같은 부분이 나타난다는 것은, 한 생물에서 다른 생물로 진화했을 가능성을 보여주기 때문이다. 이 때문에 서로 다른 시퀀스에서 공통 스트링을 찾기 위한 노력이 계속되었고, 많은 시행착오를 거쳐 현재에 이르러서는 지역 정렬, 전역 정렬 등 몇 가지 정형화된 방법과 이를 내부적으로 잘 구현한 BLAST[1], BLAT[2], PatterHunter[3] 등의 도구를 사용하여 비교적 빠른 시간 안에 정확히 원하는 조건의 공통 스트링을 찾아낼 수 있게 되었다. 이 중 suffix tree는 최근에 각광을 받기 시작한, 시퀀스의 내부 구조를 잘 드러내는 자료 구조로써, 정확하게 일치하는 문자열을 검색하는 문제에서 매우 뛰어난 성능을 보인다[4]. 본 보고서에서는 c로 구현한 suffix tree[5]의 이용하여 두 가지 조건에서 공통 스트링을 찾기 위한 실험을 수행하고, 그 성능을 평가하고자 한다. 실험 대상 문자열은 fasta 포맷의 모기 유전체 자료 6건(40∼60MB)를 사용하고, 추가적으로 한글로 구성한 파일 2건(1KB, 1MB)을 사용할 것이다.

Copyright © Graphics Application Lab. Pusan National University

2 Suffix Tree의 이해

suffix tree는 캐릭터의 스트링 매칭 문제를 정확하고 신속하게 해결할 수 있는 자료구조이다. suffix tree는 길이 n의 어떤 패턴에 대해서도 해당 패턴이 텍스트에 존재하는지 아닌지, 텍스트의 길이와 무관하게 O(n) 시간을 사용해서 답을 낼 수 있도록, 텍스트를 전처리 한다[6].

DNA 시퀀스는 패턴의 종류가 A,G,T,C로 고정되어 있고, 그 수가 적기 때문에 suffix tree를 이용하여 패턴 매칭하기에 적합한 형태이다. 그러나 일반적으로 DNA 시퀀스는 그 용량이 매우 커서 수백 MB를 남짓하는 경우가 많다. suffix tree는 정확한 패턴을 탐색하는 시간은 O(m)으로 아주 좋은 성능을 보이나, 메모리를 O(nlogn)을 사용하기 때문에 실질적으로 대용량 데이터를 처리함에 있어서는 suffix array를 사용한다. suffix tree와 array 모두 시간 복잡도와 공간 복잡도의 trade- off 관계에 있기 때문에, 유전체 데이터를 다루는 생물정보학에서는 대용량을 다룰 수 있는 suffix array를 빠르게 구축할 수 있는 연구[7]나 suffix tree의 메모리 부족 문제를 해결하기 위해 해쉬 테이블 등으로 전처리를 수행하는 연구[8]등이 이루어지고 있고, 그 밖에 문서 군집화[9] 등의 연구가 수행되고 있다.

suffix tree를 선형 시간으로 구축하는 알고리즘은 Weiner에 의해서 처음 제안되었으며, McCreight, Ukkonen은 똑같은 수행시간에 공간을 더 작게 하는 방법을 제시하였다. 특히 Ukkonen의 알고리즘은 구현과 이해가 쉽고, 온라인에서 사용가능한 선형알고리즘으로써, 대부분의 suffix tree 구축에 사용된다[10].

2.1 정의 및 특징

길이가 m인 시퀀스 S는 m개의 suffix를 가진다. 스트링 S에 대한 suffix tree는 S의 모든 suffix가 루트에서 잎 노드에 이르는 경로와 유일하게 일치하는, 루트를 가지는 트리(rooted tree) 이다. 루트가 아닌 내부 노드들은 적어도 2개 이상의 자식 노드를 가지고 있으며, 각각의 간선은 공백이 아닌 S의 부분 문자열이 라벨로 지정된다. 또한 한 노드에 연결된 간선들은 반드시 다른 문자로 시작하는 라벨을 가져야 한다. suffix tree의 특징을 정리하면 다음과 같다[11].

1. 루트와 방향성이 있는 트리이다.
2. 길이가 m인 시퀀스의 경우, 1부터 m까지의 가지를 가진다.
3. 트리의 내부 노드는 2개 이상의 자식 노드를 가진다.
4. 노드 간의 간선은 부분 문자열을 라벨로 지정한다.
5. 같은 노드를 가지는 간선들은 같은 라벨을 가질 수 없다.
6. 노드들의 라벨들은 시퀀스의 suffix 들이다.

2.2 Suffix Tree의 생성

스트링 S를 GATCCATCTT 라고 가정하자. 스트링 S로 suffix tree를 구성하는 방법은 다음과 같다.

1. S의 마지막에 $기호를 추가한다. 그 이유는 트리 구성 과정을 모두 언급한 후 설명한다.

Copyright © Graphics Application Lab. Pusan National University

2. S를 suffix의 개수만큼 나누어, 부분 스트링을 구한다. 부분 스트링 집합의 수는 잎 노드의 수와 같다. 스트링 S에 대한 부분 스트링 집합은GATCCATCTT$, ATCCATCTT$,TCCATCTT$, CCATCTT$, CATCTT$, ATCTT$, TCTT$, CTT$, TT$, T$, $ 이다.

3. 부분 스트링을 하나의 간선에 할당하고, 구성한 트리에 해당 스트링과 같은 스트링이 존재하는지 확인한다.

4. 이미 존재하는 간선과 시작 문자가 같으면, 그 간선에서 문자가 달라지는 부분에 간선을 추가해서 붙인다. 예를 들어, ATCT, ATGG라는 두 개의 문자열이 있을 경우, AT라는 공통된 라벨을 가지는 간선 밑에 각각 CT, GG 라벨을 가진 간선 두 개를 추가한다.

5. 같은 라벨을 가진 간선이 트리에 존재하면 두 노드를 suffix link로 연결한다.

과정 1에서 시퀀스의 마지막에 $ 기호를 추가하는 이유는 S의 한 suffix가 다른 suffix의 prefix와 일치할 때 처음 suffix가 잎 노드를 가지지 않고 계속될 경우를 방지하기 위함이다. 예를 들어, ABCAB 라는 문자열에서 AB는 prefix도 되고 suffix도 되기 때문에 잎 노드에서 끝나지 않는다. 이 문제를 피하기 위해 S의 마지막 문자는 암묵적으로 $ 기호를 사용하며, 이 기호는 S의 다른 부분에 나타나지 않는다고 가정한다.

그림 1은 앞서 정리한 내용을 토대로 스트링 S에 대한suffix tree를 생성한 모습이다[12]. S의 suffix 개수인 10개만큼 부분 문자열이 생성되었으며, 잎 노드의 개수 또한 그와 같음을 확인할 수 있다. 실제 구현에서는 각 부분 스트링을 모두 저장하기 위해서 소모하는 공간이 크므로, 첫글자와 마지막글자의 인덱스만을 저장한다.

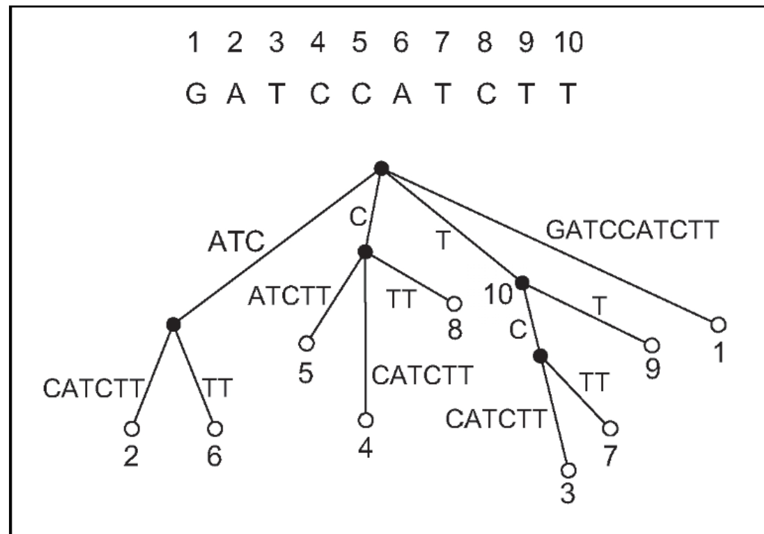

그림 1. 스트링 GATCCATCTT$에 대해 suffix tree를 구성한 모습. 같은 노드를 가지는 모든 간선들의 라벨 첫 글자가 같지 않음을 확인할 수 있으며, 부분 스트링의 개수만큼 잎 노드가 생겼음을 알 수 있다.

Copyright © Graphics Application Lab. Pusan National University

2.3 두 시퀀스의 공통 스트링 추출

suffix tree는 한 시퀀스에 존재하는 부분 스트링을 추출하는데 뛰어난 성능을 보이지만, 서로 다른 시퀀스 간에 공통 스트링을 추출하는데도 우수한 성능을 보인다. 여러 개의 시퀀스가 존재할 경우 각각에 대해 suffix tree를 구성하는 것이 아니라, 하나의 트리에 여러 시퀀스를 응집한다. 트리를 구성할 때는 한 시퀀스만으로 트리를 구성할 때와 마찬가지로 구성하되, 인덱스를 주는 부분에서 첫 자에 몇 번째 시퀀스인지 기록한다. 예를 들어 4개의 시퀀스 ATGCA, ACGCA, TAATC, TACTC에서 공통 스트링을 찾기 위한 suffix tree를 구성하는 방법은 그림 2 과 같다. 잎 노드 [α, β]는 시퀀스의 번호와 해당 시퀀스의 부분 스트링을 의미한다. 예를 들어, 잎노드 [1,2]의 경우 1번 시퀀스인 ATGCA의 2 번째 substring인 TGCA를 의미하는 것으로, 이와 같이 여러 개의 시퀀스에 대해서도 하나의 트리를 구성할 수 있다.

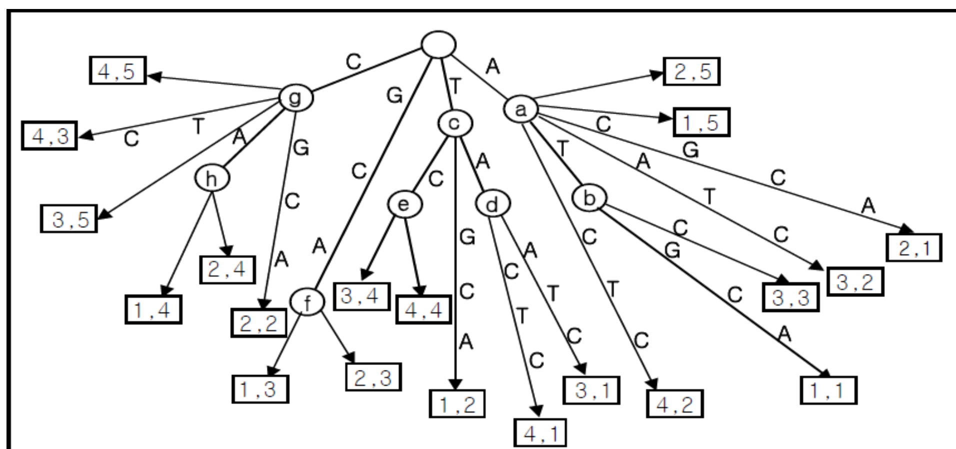

그림 2. 4개의 시퀀스(1.ATGCA, 2.ACGCA, 3.TAATC, 4.TACTC)로 구성한 Suffix Tree. 잎 노드는 [α, β]의 인덱스를 가지는데 α 는 시퀀스 번호, β 는 해당 시퀀스의 substring 번호를 나타낸다. 예를 들어, 잎노드 [1,2]의 경우 1번 시퀀스인 ATGCA의 2 번째 substring인 TGCA를 의미한다.

2.4 Suffix Tree의 구현

본 보고서에서는 suffix tree를 c 언어로 구현한 코드를 이용하였다[5]. [5]에서는 Ukkonen Algorithm으로 구현하였으나, 본 보고서에서는 McCreight Algorithm 방법을 설명하고자 한다. 두 알고리즘은 같은 시간 복잡도와 공간 복잡도를 가진다. 차이점은 McCreight Algorithm은 이미 구성한 suffix tree에 노드나 간선을 추가할 수 없다는 점이고, Ukkonen Algorithm은 추가할 수 있으며 온라인에서 구성할 수 있다는 것이다. 따라서 비교적 이해하기 쉬운 McCreight 방법을 통해 suffix tree를 구현하는 방법에 대해 알아보고자 한다.
알고리즘 1은 McCreight Algorithm을 사용하여 suffix tree를 구성한 Pseudo Code이다. suffix tree의 구성 과정은 크게 slow scan과 fast scan으로 나눌 수 있다. slow scan은 이미 구성한 트리의 간선 라벨과, 추가하려는 부분 문자열이 같은지 character by character 로 하나씩 검사하는 방법이다. fast

Copyright © Graphics Application Lab. Pusan National University

scan 을 부분 문자열 TCTT와 TT를 예로 들어 설명하면, 두 문자열에서 prefix인 T가 같음을 알 수 있는데, 이러한 T를 LCP(Longest Common Prefix)라 한다. LCP가 이미 구성한 트리에 존재할 경우 해당 간선 아래로 건너뛰는 것을 fast scan이라 정의한다. 이 두 가지 scan 방법을 통해 그림 1과 같은 suffix tree를 구성할 수 있다.

Algorithm 1 McCreight Algorithm 방법을 사용하여 suffix tree를 구성한 Pseudo Code.

```
1.   Construct Tree for x[1..n]
2.   P = 0
3.   for ( i = 1 to n do )
4.       if (head(i) == root then
5.               head(i+1) = slowscan(root, s(tail(i)))
6.               add i+1 and head(i+1) as node if necessary
7.               continue
8.       u = parent(head(i)); v = label(u, head(i))
9.       if u != root then w = fastscan(s(u),v)
10.       else              w = fastscan(root,v[2...|v|])
11.       if w is an edge then
12.               add a node for w
13.               head(i+1) w
14.       else if w is a node then
15.               head(i+1) = slowscan(w, tail(i))
16.               add head(i+1) as node if necessary
17.       s(head(i)) = w
18.       add leaf i+1 and edge between head(i+1) and i+1
```

알고리즘 1의 17, 18번째 코드는 간선의 라벨이 같은 노드끼리 연결하는 부분으로, 그림 1 에서 제외된, suffix link로 연결하는 부분이다. 간선의 라벨이 같아도 트리에서의 위치가 다르면 간선을 새로 추가해야만 suffix tree를 구성할 수 있는데, 이 경우 suffix tree의 구성 시간은 O(m)으로 선형 시간이 아니게 된다. 따라서 같은 간선을 suffix link를 통해 연결함으로써, tree에서 중복 간선을 제거할 수 있어 공간 복잡도도 줄일 수 있고, suffix tree의 구성도 선형 시간에 완료할 수 있다. 트리를 구성하고 난 이후에는 비교 대상인 스트링 S의 길이인 n과 무관하게 검색하고자 하는 스트링 x의 길이 m 만큼을 탐색 시간으로 소요한다.

3 Suffix Tree의 시간/공간 복잡도 실험

3.1 입력 파일의 특성

suffix tree의 성능을 평가하기 위한 입력 자료로 fasta 포맷의 모기 유전체 파일 2개모기 유전체에서 용량을 축소한 더미 유전체 파일 5개, 21세기 세종계획의 현대 구어 원시 말뭉치 1개, 한글 상호 명으로 구성한 파일 1개를 사용하였다. 입력 파일의 용량에 대한 자세한 정보는 표 1과 같다.

Copyright © Graphics Application Lab. Pusan National University

3.2 실험 방법

실험한 환경은 Windows 7 64bit OS에 RAM 4.00GB, gcc compiler version 3.4.2 이다. suffix tree의 탐색 시간은 c의 time 함수를 사용하여 측정하였다. 실험은 총 세 가지로 나누어 진행하였다. (1) 표 1 에서 사용한 9가지 종류의 파일을 사용하여 한 가지 파일에 대해서 패턴의 길이를 달리하여 실험하고, (2) 여러 가지 파일에 대해 같은 길이의 패턴을 사용하여 탐색 시간을 측정하였다. 또한 (3) 파일에 존재하지 않지만, 존재하는 패턴과 아주 유사한 패턴을 검색어로 입력하여 이를 탐색할 수 있는지에 대해 검사하였다. (1)과 (2)의 실험 목적은 대상 텍스트의 길이 n이 아닌, 패턴의 길이 m에 비례하는 O(m) time에 suffix tree가 구현되는지에 대한 여부를 확인하기 위함이다. 실험 (3)은 suffix tree가 비슷한 스트링의 검색에 취약할 것이라는 가정이 맞는지 확인하는 것이 그 목적이다.

파일 종류	파일 명	파일 용량[KB]
모기 유전체1	chr2R.fa	62,481
모기 유전체2	chr2L.fa	48,605
dummy 유전체1	f4.fa	29,419
dummy 유전체2	f1.fa	26,211
dummy 유전체3	f3.fa	20,220
dummy 유전체4	lagan1.fasta	15,589
dummy 유전체5	f2.fa	10,110
한글 말뭉치1	korean2.txt	1,030
한글 상호 명1	korean.txt	1

표 1. suffix tree의 성능을 평가하기 위한 입력 자료. 6MB ∼1KB 범위의 용량을 사용하며, DNA와 한글을 입력 신호로 사용한다.

3.3 실험 결과 및 분석

3.1의 9가지 파일 중 한글 상호 명1 파일을 제외한 나머지에 대해서는 각각 패턴의 길이가 50자, 5자인 경우에 대해 실험하고 실험에 소요한 시간을 측정하여 표 2로 나타내었다. 소요 시간은 suffix tree를 구성하는 데 사용한 시간과 탐색 시간을 포함한다.

표 2에서 확인할 수 있듯이, 용량이 큰 모기 유전체 데이터는 suffix tree를 구성하던 중 메모리 부족으로 탐색을 수행하지 못했음을 알 수 있다. 이는 suffix tree 구성을 선형 시간에 완료하기 위해서 같은 prefix는suffix link를 사용해 연결하기 때문이다. 일각에서는 이와 같은 suffix tree의 공간 복잡도를 낮추기 위한 연구[13]도 진행하였다. 29MB 이상의 파일은 메모리 부족으로 정확한 실험 결과를 얻지 못하였다. 26MB 이하의 파일에서는 패턴 길이에 비례해서 탐색 시간이 증가하는 것을 비교적 명확히 확인할 수 있었다. 용량이 증가할수록 패턴 길이와 소요 시간이 크게 비례하지 않는 것을 확인할 수 있는데, 이는 트리를 구성하는데 소요한 시간이 커서 패턴의 길이에 따른 탐색 시간이 전체 소요 시간에 영향을 주지 못했기 때문인 것으로 생각된다. 한글 말뭉치의 경우 전체 용량이 1KB로 너무 적어, 시간을 측정한 time 함수의 오차 범위 내에 두 패턴에 따른 소요 시간의 차이가 존재하므로,

Copyright © Graphics Application Lab. Pusan National University

파일 종류	소요 시간[sec]	
	50자 패턴	5자 패턴
모기 유전체1	out of memory	out of memory
모기 유전체2	out of memory	out of memory
dummy 유전체1	out of memory	out of memory
dummy 유전체2	35.16200	35.00600
dummy 유전체3	21.74600	21.69900
dummy 유전체4	17.42500	16.95700
dummy 유전체5	12.01200	11.74600
한글 말뭉치1	0.73300	0.71700

표 2. 각 파일을 50자, 5자 패턴에 대해 suffix tree를 구성하고 패턴 탐색을 수행하는데 소요한 시간 측정 실험 결과. 패턴 길이와 파일 용량에 소요시간이 비례함을 알 수 있다.

패턴에 길이에 따른 suffix tree의 탐색 시간을 확인하기가 어렵다. 대용량의 파일에서도 suffix tree의 패턴 탐색 시간이 패턴의 길이에 영향을 받음을 확인하기 위해서는, 트리 구성 시간을 높이더라도 suffix link를 제거하여 공간 복잡도를 낮추어 실험해야 할 것으로 생각된다. suffix tree의 유사 패턴 탐색에 대한 성능을 평가하기 위해서 한글 상호 명1 파일에 존재하는 패턴과, 존재하지 않지만 유사한 패턴을 탐색하였다. 실험 결과는 표 3과 같다.

정확한 패턴	변형 패턴	변형 유형	탐색 성공 여부
오븐에꾸운닭	오븐에빠진닭	수정	실패
교촌치킨	교촌발치킨	삽입	실패
비에이씨치킨	비에씨치킨	삭제	실패
맥시칸	멕시칸	수정	실패
또래오래	또래오래	없음	성공

표 3. suffix tree의 성능을 평가하기 위한 입력 자료. 6MB ∼1KB 범위의 용량을 사용하며, DNA와 한글을 입력 신호로 사용한다.

패턴을 한 단어만 수정, 삽입, 삭제하여 변형하여 이를 검색어로 사용하였을 경우, 그림 3과 표 3에서 확인할 수 있듯이 suffix tree는 아주 나쁜 성능을 보였다. 이를 통해 suffix tree는 정확한 단어의 매칭에만 뛰어난 성능을 보이고, 유사한 단어의 탐색에 대해서는 매우 취약한 것을 알 수 있다. 지금까지 확인한 suffix tree의 성능 평가 실험을 정리한 결과는 다음과 같다.

1. Suffix tree는 패턴의 길이 n에 비례하는 O(n) time에 구현된다.
2. Suffix tree를 이용하면 정확한 단어의 매칭 문제에는 뛰어난 성능을 보이지만, 유사한 단어를 탐색하는 문제에는 취약하다.

Copyright © Graphics Application Lab. Pusan National University

```
F:₩aGALab_SeonYeong₩bMaster1₩생물정보학특강₩Mid-Term Technical Report>
suffix_tree f korean.txt 오븐에빠진닭
Constructing tree.....Done.

Results:      String is not a substring.

0.000000sec

F:₩aGALab_SeonYeong₩bMaster1₩생물정보학특강₩Mid-Term Technical Report>
suffix_tree f korean.txt 교촌발치킨
Constructing tree.....Done.

Results:      String is not a substring.

0.000000sec

F:₩aGALab_SeonYeong₩bMaster1₩생물정보학특강₩Mid-Term Technical Report>
suffix_tree f korean.txt 비에씨치킨
Constructing tree.....Done.

Results:      String is not a substring.

0.000000sec

F:₩aGALab_SeonYeong₩bMaster1₩생물정보학특강₩Mid-Term Technical Report>
suffix_tree f korean.txt 멕시칸
Constructing tree.....Done.

Results:      String is not a substring.

0.000000sec

F:₩aGALab_SeonYeong₩bMaster1₩생물정보학특강₩Mid-Term Technical Report>
suffix_tree f korean.txt 또래오래
Constructing tree.....Done.

Results:      Substring exists in position 379.

0.000000sec
```

그림 3. 표 3의 변형 패턴 실험 결과. 정확한 단어가 아니면 스트링이 존재하지 않는다고 판단하는 것을 알 수 있다.

4 결론 및 추후연구

본 보고서에서는 suffix tree의 동작 원리에 대해 알아보고, 실제로 suffix tree를 사용하여 다양한 signal에 대해 실험해보았다. suffix tree는 시퀀스의 길이 n, 패턴의 길이 m에 대하여 트리 구성을 O(n+m), 패턴 탐색을 O(n) 시간에 완료할 수 있는 뛰어난 성능을 보이는 자료구조였다. suffix tree로 fasta 포맷의 모기 유전체 자료, 더미 유전체 자료, 한글 말뭉치, 한글 상호 명 파일에 대해 패턴 탐색 실험을 수행한 결과, 29MB 이상의 파일에서는 메모리 부족으로 인해 성능 평가를 할 수 없었으나, 그 이하 용량의 파일에 대해서는 패턴의 길이에 비례하여 탐색 시간이 증가함을 확인할 수 있었다. 한글 상호명으로 구성한 파일에서는 정확한 상호명일 경우 선형 시간에 탐색을 마치고 바른 결과를 도출했으나, 한 글자라도 변형된 스트링의 경우에는 유사한 스트링을 탐색하지 못하는 것을 확인하였다. 즉, 유사한 단어의 검색에 대해서는 매우 취약함을 알 수 있었다. 따라서 suffix tree는 키워드를 정확하고 신속하게 찾을 필요성이 있을 경우, 대용량이 아닌 파일에서 특정 패턴을 찾아야 하는 경우에 유용하게 사용할 수 있을 것으로 기대한다. 실험 도중 suffix tree의 구성 시간을 줄이기 위해서 사용하는 suffix link로 인한 메모리의 소비가 심하여, 대용량의 파일에 대해서는 메모리 부족으로 성능 평가를 제대로 할 수 없었는데, 추후 suffix tree의 공간 복잡도를 축소할 수 있는 방법에 대해서 연구하여 대용량 파일에서도 선형 시간에 suffix tree를 사용할 수 있도록 할 계획이다.

Copyright © Graphics Application Lab. Pusan National University

참고 문헌

1. S.F. Altschul, W. Gish, W. Miller, E. Myers, and D.J. Lipman, "Blast: Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, pp. 403–410, 1990.
2. W.J. Kent, "Blat: The blast-like alignment tool," *Genome Res.*, vol. 12, no. 4, pp. 656–664, 2002.
3. B. Ma, J. Tromp, and M. Li, "Patternhunter: faster and more sensitive homology search," *Bioinformatics*, vol. 18, no. 3, 2002.
4. Horatiu Mocian, Ed., *Text Mining with Suffix Trees*, Imperial college London.
5. Shlomo Yona, "Ansi c implementation of a suffix tree," http://mila.cs.technion.ac.il/yona/suffix_tree/.
6. Neil C. Jones and Pavel A. Pevzner, Eds., *An Introduction to Bioinformatics Algorithms*, The MIT Press.
7. 조준하, 김남희, 권기룡, and 김동규, "Dna 스트링에 대하여 써픽스 배열을 구축하는 빠른 알고리즘," *정보과학회논문지*, vol. 34, no. 7, pp. 319–326, 2007.
8. 송혜주 and 박영호, "효과적인 dna 검색을 위한 해쉬 기반의 서픽스 트리(suffix tree) 방안," 2008, vol. 35, pp. 172–175.
9. 조윤호 and 이상근, "공통 phrase의 관계 그래프와 suffix tree 문서 모델을 이용한 문서 군집화 기법," *한국콘텐츠학회 논문지*, vol. 9, no. 2, pp. 142–151, 2009.
10. Dan Gusfield, Ed., *Algorithms on Strings, Trees, and Sequences*, Cambridge.
11. 이성근, 허보경, 안대명, and 황규석, "Suffix tree를 이용한 dna sequences의 clustering에 관한 연구," *abc*, vol. 8, no. 2, pp. 2861–2864, 2002.
12. Kun-Mao Chao and Louxin Zhang, Eds., *Sequence Comparison*, Springer.
13. 송혜주 and 박영호, "효과적인 dna 검색을 위한 해쉬 기반의 서픽스 트리(suffix tree) 방안," 2008.

Copyright © Graphics Application Lab. Pusan National University