공간 효율적인 DNA 시퀀스 인덱싱 방안
(A Space Efficient Indexing Technique for DNA Sequences)

송 혜 주 † 박 영 호 †† 노 웅 기 †††
(Hye-Ju Song) (Young-Ho Park) (Woong-Kee Loh)

요 약 서픽스 트리는 공통의 프리픽스의 빈도수가 높을 때 효과적인 알고리즘으로, 한정된 문자로만 구성된 DNA 유사성 검색을 위한 연구에서 널리 활용되고 있다. 그러나, 서픽스 트리는 인덱스 특성 상 메모리 공간을 많이 차지하며, 트리의 분할 시 DNA 시퀀스의 비율로 인한 쏠림현상이 발생한다는 문제점을 가진다. 따라서, 본 논문에서는 공통의 프리픽스를 가지는 가변길이의 파티셔닝 방법으로 합병하지 않는 인덱싱 방안인 SENoM을 제안한다. SENoM은 전체 시퀀스에서 공통의 프리픽스를 가지는 서픽스들의 발생 빈도수가 임계치 이하인 경우 디스크에 저장하고, 임계치 이상인 경우 임계치 이하가 될 때까지 프리픽스를 확장한다. 모든 파티션은 서브트리로 구축한 후 디스크에 저장하며, 질의처리를 위해, 구축된 파티션의 프리픽스를 서픽스로 가지는 트리를 구축한다. 제안하는 기법은 복잡한 합병과정을 제거하고, 많은 파티션 발생으로 인한 디스크 I/O 발생을 줄인다. 실험을 통해, SENoM이 Trellis 알고리즘에 비해 메모리 사용량을 약 35%, 인덱스 크기를 약 20% 감소시켰음을 보인다. 또한, 질의길이가 긴 경우에도 프리픽스 트리를 이용하여 효과적인 질의처리가 가능함을 보인다.

키워드 : 서픽스 트리, 가변길이 프리픽스

Abstract Suffix trees are widely used in similar sequence matching for DNA. They have several problems such as time consuming, large space usages of disks and memories and data skew, since DNA sequences are very large and do not fit in the main memory. Thus, in the paper, we present a space efficient indexing method called SENoM, allowing us to build trees without merging phases for the partitioned sub trees. The proposed method is constructed in two phases. In the first phase, we partition the suffixes of the input string based on a common variable-length prefix till the number of suffixes is smaller than a threshold. In the second phase, we construct a sub tree based on the disk using the suffix sets, and then write it to the disk. The proposed method, SENoM eliminates complex merging phases. We show experimentally that proposed method is effective as bellows. SENoM reduces the disk usage less than 35% and reduces the memory usage less than 20% compared with TRELLIS algorithm. SENoM is available to query efficiently using the prefix tree even when the length of query sequence is large.

Key words : Suffix Tree, Variable-length prefix

· 본 연구는 숙명여자대학교 SRC여성질환연구센터 특별연구비 지원으로 수행되었음(2008)

† 학생회원 : 숙명여자대학교 멀티미디어과학과
thdgPwn@gmail.com
†† 종신회원 : 숙명여자대학교 멀티미디어과학과 교수
yhpark@sm.ac.kr
(Corresponding author임)
††† 정 회 원 : 성결대학교 멀티미디어학부 교수
woong@sungkyul.ac.kr

논문접수 : 2008년 12월 18일
심사완료 : 2009년 10월 8일

정보과학회논문지: 데이타베이스 제36권 제6호(2009.12)

1. 서 론

서열 유사성 검색은 생물 정보학 분야에서 유전자 서열의 기능을 유추하거나 관련 있는 서열들 간의 진화관계를 예측하기 위해 중요한 연구 분야이다. 국·내 외에서는 최근 몇 년을 거치면서 서열 유사성 검색을 위한 정교한 생물학적 검색 엔진들이 만들어졌으며, 현재까지도 활발히 연구되고 있다.

가장 널리 쓰이는 유전자 검색 알고리즘으로 BLAST (Basic Local Alignment Search Tool)[1]가 대표적이다. BLAST는 미국 국립 생물 정보 센터(National Center for Biotechnology Information, NCBI)에서는 보유하고 있는 데이터에 대한 종합검색시스템으로, 핵산 염기서열, 단백질 아미노산 서열, 단백질 3차 구조, 유전자

(genome) 정보, 분류학적 정보, 문헌 데이터베이스 등에 접근하여 결과를 추출한다[3]. BLAST는 전체 데이터베이스와 질의서열을 비교하여 서열간의 최적의 정합을 찾는 워터만 알고리즘(Smith-Waterman)의 검색 속도를 향상시켜, 높은 상동성을 가진 서열들을 빠른 시간 내에 찾을 수 있다.

그러나, 기존의 유전자 검색 시스템과 연구들은 다음과 같은 문제점이 존재한다. 먼저, BLAST의 경우, 관련 데이터베이스가 주기억 장치에 적재되어야 하며, 검색속도가 데이터베이스 크기에 비례하여 증가한다. 또한 워드의 고정 길이에 따라 검색결과의 정확도가 영향을 받는 점 등이 문제점으로 지적된다. 이를 해결하기 위해, 많은 연구에서 서픽스 트리(Suffix Tree)를 이용한 인덱싱 방법을 활용하고 있다.

서픽스 트리는 공통의 프리픽스(Prefix)를 가지는 서픽스의 빈도수가 높을 때 효과적인 알고리즘으로, DNA 시퀀스 검색을 위한 연구에서 널리 활용되고 있다.

Ela Hunt가 제안한 Phased Tree 알고리즘[4]은 서픽스 트리의 메모리 문제를 해결하기 위한 방법으로, 트리의 디스크 분할을 통해 대규모의 서픽스 트리를 구성한다. 제안된 방안은 서픽스 링크(Suffix Link)를 제거하여 메모리 병목현상(Memory Bottleneck)의 발생을 없애고, 참조국부성을 개선한다. 그러나, Phased Tree 알고리즘에서는 같은 프리픽스를 포함하는 모든 서픽스(Suffix)들을 PJama(JAVA 기반의 객체 저장소) 기반의 디스크의 서브트리(SubTree)로 삽입하므로, 트리를 구축하는 동안 디스크의 랜덤 엑세스가 많이 발생한다는 문제점을 가진다.

Phased Tree 알고리즘의 문제점을 해결하기 위해 제안된 방안이 TDD(Top-Down Disk-Based) 알고리즘이다. TDD 알고리즘은 Phased Tree 알고리즘과 같이 서픽스 링크를 사용하지 않으며, Wotdeager 알고리즘 기반의 서픽스 트리 구축 방안과 분할을 제안한다. 그러나, 전체 시퀀스의 크기가 커질수록 각 서픽스에 대한 전처리 비용이 추가적으로 발생하며, 일부 분할의 경우 쏠림 현상(Data Skew)[8]이 발생한다.

TRELLIS(External Suffix TRee with Suffix Links for Indexing Genome-scaLe Sequence) 알고리즘[14]에서는 쏠림현상을 해결하기 위해 가변길이 방식의 서픽스트리를 생성한다. TRELLIS 알고리즘은 가변의 프리픽스를 기반으로 파티션을 분할한 후, 합병함으로써 대용량의 DNA 시퀀스를 빠른 시간 내에 메모리에 구축할 수 있다. 그러나, 트리구축 시 많은 메모리와 디스크 공간이 요구되며, 공통의 프리픽스를 가진 서브트리를 합병하기 위한 디스크 I/O가 발생하게 된다.

따라서, 본 논문에서는 합병하지 않는 Wotdeager 기반의 공간 효율적인 SENoM(Space Efficient No Merge) 인덱싱 방안을 제안한다. SENoM은 다음과 같이 세 단계로 진행된다. 첫째, 전체 시퀀스를 스캔 하면서 공통의 프리픽스를 가지는 서픽스 간 파티션을 생성한다. 이때, 공통의 프리픽스를 갖는 서픽스들의 발생 빈도수가 임계치(Thredhold) 이상인 경우, 프리픽스를 확장하여 재분할을 진행한다. 둘째, 서픽스들의 발생 빈도수가 임계치 이하인 파티션에 대하여 각 서픽스를 서브트리로 구축한다. 셋째, 서브트리로 구축된 파티션의 프리픽스를 서픽스로 가지는 트리를 메모리상에 구축한다. 이는 질의처리를 위한 단계로써, 서브트리의 상위트리를 통합하여 서픽스 트리 상에서 검색할 수 있다.

제안하는 SENoM은 모든 서브트리가 고유한 프리픽스를 가짐으로써 복잡한 합병단계를 제거하고, TRELLIS 알고리즘에 비해 메모리와 디스크 공간 요구량을 줄일 수 있다는 장점을 가진다. DNA와 같이 방대한 데이터의 경우, 효율적 메모리 관리뿐 만 아니라, 한정된 메모리만으로 인덱스를 구축하는 것도 중요하기 때문이다.

본 논문은 다음과 같은 공헌을 제시한다.

- 본 논문에서는 대용량의 DNA 시퀀스를 인덱싱하기 위한 공간 효율적인 서픽스 트리 인덱싱 방안인 SENoM을 제안한다.
- SENoM은 합병단계를 제거하여, DNA 시퀀스의 트리 구축 시 문제가 되는 메모리와 디스크 사용량을 TRELLIS 알고리즘보다 줄인다.
- 실험을 통해, SENoM은 TRELLIS와 동등한 속도로 트리를 구축하며, 질의가 긴 경우에도 빠른 질의처리가 가능함을 보인다.

본 논문의 구성은 다음과 같다. 2장에서는 DNA서열의 유사성을 검색하는 기존의 알고리즘과 Phased Tree, TDD, TRELLIS 등을 설명하고, 3장에서는 합병하지 않는 디스크 기반의 SENoM 인덱싱 방안에 대하여 설명한다. 4장에서는 구축한 인덱스에서의 질의처리방안에 대해 설명하고, 5장에서는 TRELLIS 알고리즘과 제안하는 SENoM을 비교, 분석한 실험결과를 보인다. 마지막으로 6장에서는 결론을 내린다.

2. 관련연구

본 장에서는 서열 유사성 검색에 널리 쓰이는 시스템인 BLAST와 서픽스 트리를 활용한 Phased Tree, TDD 알고리즘을 소개한다.

2.1 BLAST 알고리즘

생물 정보학 연구에서 가장 널리 사용되고 있는 BLAST[3]는 서열 데이터베이스 내에서 유사성을 지닌 아미노산이나 염기서열을 찾는 휴리스틱스(Heuristics) 기반의 알고리즘이다. BLAST 알고리즘으로 DNA 서

열의 유사성을 검색하기 위해서는 다음과 같은 과정을 수행한다. 먼저 DNA 시퀀스 S에서 길이가 |W|로 고정된 슬라이딩 윈도우를 위치시켜 서열의 목록을 추출한다. 여기서 추출된 서열의 개수는 S-|W|+1이다.

두 번째로, 데이터베이스를 스캔하여 앞에서 추출한 목록들과 정확히 일치하는 부분서열을 찾아낸다. 마지막으로 일치하는 부분서열들을 양쪽 방향으로 확장하여 비교하고, 일정 이상의 HSP(High-scoring Segment Pair)을 가지는 서열들을 추출한다. BLAST는 워터만 알고리즘과 FASTA 알고리즘의 성능을 향상시켜 유사한 서열들을 빠른 시간 내에 찾아낸다[3]. 그러나, 서열 데이터베이스와 윈도우의 크기가 커지면 검색속도가 증가하며, 결과 서열의 정확도가 떨어지는 단점을 가진다. 따라서, DNA 유사성 검색 시 검색속도와 정확도를 개선할 인덱싱 연구들이 요구된다.

2.2 Phased Tree 알고리즘

Ela Hunt가 제안한 Phased Tree 알고리즘[4,15]은 서픽스 트리의 메모리 문제를 해결하기 위해 제안된 방법으로, 트리의 디스크 분할을 통해 대규모의 접미어 트리를 구성한다.

제안된 Phased Tree 알고리즘에서는 먼저, 주어진 시퀀스 에서 고정된 길이의 프리픽스를 계산하는데, 프리픽스의 길이는 공통의 프리픽스를 가지는 서픽스들로 구성된 서브트리가 메인 메모리에 최적화 될 수 있도록 고정한다. 다음으로, 고정된 프리픽스로 시작하는 각 서픽스는 디스크 기반의 서픽스에 삽입된다.

제안된 방안은 서픽스 링크를 제거하여 메모리 병목현상(Memory Bottleneck)의 발생을 없애고, 참조 국부성(Locality of Reference)을 개선한다[4]. 그러나, 같은 프리픽스를 포함하는 모든 서픽스들은 PJama(JAVA 기반의 객체 저장소)기반의 디스크의 서브트리로 삽입되므로, 트리를 구축하는 동안 디스크의 랜덤 엑세스가 많이 발생한다는 문제점을 가진다. 또한, 실제 DNA 시퀀스는 일정한 비율로 구성되어 있지 않기 때문에 쏠림현상(Data Skew)이 발생하며[8], 일부 분할의 경우 메인 메모리의 크기를 초과할 수도 있다는 문제점을 가진다. 이를 해결하기 위해 bin-Packing 알고리즘을 적용하지만, 이론적으로만 제안되었으며, 고정된 프리픽스의 길이가 길어지면 bin-Packing으로 인한 비용이 많이 발생한다[8].

2.3 Top-Down 서픽스 트리 구축 알고리즘

Phased Tree 알고리즘의 문제를 해결하기 위해 제안된 알고리즘은 Wotdeager 알고리즘 기반의 Top-Down 서픽스 트리 구축 알고리즘[9,10]이다. Wotdeager 알고리즘은 처음 엑세스 할 때 모든 서브트리를 구축하며, 효과적인 캐쉬 이용을 위한 구체적인 버퍼링 전략을 포함한다[12]. 이러한 접근은 전체 시퀀스에 비해 쿼리가 짧을 때 효과적이나, 쿼리가 길 때에는 전체트리가 메모리에 할당되어야 하므로, 성능이 좋지 않다.

TDD 알고리즘[9]은 Top-Down 방식을 개선한 알고리즘으로, Phased Tree 알고리즘과 같이 서픽스 링크를 사용하지 않으며, Wotdeager 알고리즘 기반의 서픽스 트리 구축 방안과 분할을 제안한다. TDD 알고리즘에서는 TOP-Q 접근[16]의 성능분석을 통해, TDD의 성능을 향상시킨 ST-Merge 기법을 제시한다.

ST-Merge 방법에서는 전체 스트링을 연속된 서브스트링으로 분할한 후, TDD방식을 이용하여 서픽스 트리를 구축한다. 이후, 구축된 서브트리들을 함께 합병하여 완성된 트리를 구축한다. TDD는 메모리 효율성이 좋으나, 전체 시퀀스의 크기가 커질수록 각 서픽스에 대한 전처리 비용이 추가적으로 발생하며, 고정길이의 프리픽스 분할로 인한 쏠림현상이 발생한다.

2.4 TRELLIS 알고리즘

TDD 알고리즘의 문제점인 쏠림 현상을 해결하기 위해 새롭게 제안된 방법이 TRELLIS 알고리즘[14]이다. TRELLIS 알고리즘은, 전체 시퀀스를 스캔하여 발생 가능한 가변길이의 프리픽스 집합을 구성하고, 시퀀스를 순차적으로 분할하여 임계치를 초과하지 않는 Ukkonen 알고리즘 방식[13]의 서브트리를 만든다. 다음으로, 각 서브트리에서 공통의 프리픽스를 가지는 노드 간 분할하게 된다. TRELLIS 알고리즘은 전체 시퀀스를 분할하여 트리를 구축하고, 이에 대한 서브트리를 구성하므로, 전체 시퀀스가 아닌 서브트리에 대한 시퀀스를 메모리에 한번에 올릴 수 있다[14]. 그러나, 서브 트리를 다시 분할함으로써 발생하는 디스크 I/O와 서브 트리 간 공통의 프리픽스를 가진 파티션을 생성하기 위한 합병 비용이 발생하며, 메모리와 디스크 공간의 오버헤드 크다는 문제점이 존재한다.

따라서, 본 논문에서는 이와 같은 문제점을 해결하고, 메모리와 디스크 공간의 오버헤드를 줄이는 합병 없는 가변길이 방식의 서픽스 트리 구축방식을 제안한다.

3. SENoM 인덱싱 방안

본 장에서는 DNA 서열 데이터베이스에서 질의서열과 유사한 서열을 찾기 위해, 공간 효율적인 서픽스 트리 구축 방안인 SENoM을 제안한다. 3.1절에서는 가변길이의 프리픽스 분할 방안을 소개하고, 3.2절에서는 트리의 노드를 저장하는 방안을 설명한다. 3.3절에서는 합병하지 않는 파티션 분할 방식을 자세히 소개한다.

3.1 가변길이 서픽스 트리 구축 방안

본 연구에서 서픽스 트리의 구축 알고리즘은 다음과 같은 기본적인 접근을 기반으로 한다. 첫 번째로, 서픽

스 링크(Suffix Link)를 제거한다. 서픽스 링크는 각 내부노드에 존재하며, 같은 프리픽스(Prefix)를 가지는 서픽스를 인덱스하는 트리 노드를 포인팅 하고 있다. 그러나, 서픽스 링크는 기존 노드들의 프리픽스와 일치하는 새로운 노드가 생성되면, 이미 구축된 트리를 변경하므로, 메모리 병목(Memory Bottleneck) 현상이 발생 가능하며, 서픽스를 가리키는 포인터 정보를 저장해야 한다는 점에서 메모리 비효율적이다. 따라서, 본 논문에서는 서픽스 링크(Suffix Link)를 사용하지 않는다.

두 번째로는 프리픽스의 길이가 고정되어있지 않는 가변길이 분할방식을 채택한다. 가변길이 분할방식은 TRELLIS 알고리즘에서 처음 제안된 기법으로, 쏠림현상을 해결한다. 쏠림현상이란 고정길이의 프리픽스로 스트링을 분할하였을 때, 각 프리픽스를 공유하는 서픽스들의 발생 빈도수가 일정하지 않아 발생하는 문제이다. 예를 들어, 인간유전자의 경우 LCP(Longest Common Prefix)의 길이가 1일 때, 각각의 프리픽스 A, C, T, G는 약 30%, 20%, 20%, 30%의 비율로 분할되므로, 일부 분할된 서브 트리의 경우 크기가 비대해 질 수 있다[8]. 또한, LCP 값이 클 경우, 많은 파티션이 생성되어 리소스 부하의 원인이 되며, LCP의 값이 작은 경우에는 메모리보다 큰 서브 서픽스 트리가 생성되어 추가적인 디스크 I/O를 발생시킨다.

따라서, 제안하는 SENoM은 Wotdeager 알고리즘 기반의 가변길이 프리픽스 분할 단계와 서브 서픽스 트리 구축 단계로 구성되며, 그림 1과 같은 순서로 진행된다.

표 1은 4장에서 알고리즘 설명을 위하여 사용된 기호들의 정의이다.

표 1 기호 정의

Pprefix	공통 프리픽스를 갖는 서픽스들의 집합
C(Pprefix)	Pprefix 서픽스들의 빈도발생수
|Pprefix|	공통의 프리픽스의 길이
Sizeof(Input)	전체 데이터 크기
Mavailable	전체 메모리 가용량
t	임계치

Step 1. 전체 시퀀스를 스캔 하면서 *|Pprefix|* = 1의 공통의 프리픽스 갖는 파티션으로 분할하는 단계이다. 각 파티션은 A, C, G, T로 구성된 각각의 프리픽스를 가지는 네 개의 파티션으로 분할된다.

Step 2. 모든 파티션 *Pprefix*가 식 (1)을 만족할 때까지 파티션을 분할한다. 즉, 공통의 프리픽스를 갖는 서픽스들의 발생빈도수 *C(Pprefix)*가 임계치(Thredhold) *t*이하인 경우, 파티션으로 분할되어 디스크에 저장되며, 발생빈도수가 임계치 이상인 경우, 해당 파티션은 큐에 삽입된다. 모든 파티션이 임계치 이하일 때까지 프리픽스를 확장하여 파티션을 분할하며, 분할된 파티션은 큐에서 삭제된다.

$$C(Pprefix) \leq t \quad (1)$$

파티션 구축을 위한 비용은 식 (2)와 같다. *n*은 서픽스에서 오프셋을 의미하며, *Prefix n*은 *n*번째 위치의 문자열을 의미한다. *max Prefix*는 파티션이 임계치 이하의 발생빈도수를 만족하는 최대 프리픽스의 길이를 의미한다.

$$O(\sum_{n=0}^{max\ prefix} C(Pprefix\ n)) \quad (2)$$

임계치 *t*는 서픽스들로 서브트리를 구성하였을 때 요구되는 메모리 양이 현재 가용메모리 Mavailable를 초과하지 않는 최대치로 계산된다. 일반적으로 DNA 시퀀스에서 리프노드의 수는 서픽스의 수와 동일하고, 내부노드의 수는 리프노드 * 0.75 수준이다[14]. 본 논문에서 리프노드와 내부노드는 각각 12byte의 메모리가 요구되므로, 다음 식 (3)과 같이 예측되는 노드의 수를 계산할 수 있다.

$$12(C(Pprefix)*1.75) + Sizeof(Input)/4 \leq Mavailable \quad (3)$$

서픽스의 발생빈도수를 *C(Pprefix)*, 인덱싱하는 전체 데이터 크기를 *Sizeof(Input)*라고 할 때, 임계치 *t*는 식 (3)을 만족하는 최대 *C(Pprefix)* 값과 같다. 식 (3)에서 전체 데이터를 4로 나누는 이유는 전체 문자열을 비트 단위로 변환하여 메모리에 할당하기 때문이다.

Step 3. *Pprefix*가 식 (1)을 만족하는 파티션에 대해 모든 서픽스를 서브트리 *Tsub*로 구축한다. 구축된 서브트리의 크기는 *Mavailable*를 초과하지 않는 최대치이다. *Tsub*에서 노드 삽입은 루트 노드로부터 하위노드를 비교하여 라벨간 미스매치가 일어날 때까지 경로를 따라가거나, 삽입된 서픽스가 모두 매치될 때까지 진행된다. 전자의 경우, 마지막 매치된 오프셋 *i* 이후의 서픽스가 새로운 리프노드로 삽입되어 *i*번째 라벨까지 인덱싱하는 노드를 부모 노드(Parents Node)로 가지며, *i*부터 나머지 부분의 라벨을 인덱싱하는 새로운 형제노드(Sibling Node)를 생성한다. 후자의 경우에는 서픽스와 전체 매치된 노드 *n*의 자식노드로 생성되거나, *n*의 자식노드의 형제노드로 추가된다.

각 파티션은 모두 다른 프리픽스를 공유하므로 임의의 파티션이 서브트리로 구축된 후에는 다른 파티션이 서브트리로 구축되는 과정에서 다시 메모리에 로드되지 않아도 된다. 따라서, 파티션으로부터 생성된 서브트리는 디스크에 바로 기록된다. 디스크에서의 트리노드 저장구조는 4장에서 자세히 언급하도록 한다.

Step 4. 마지막 단계에서는 모든 파티션의 프리픽스에 대한 트리 *Tpre*를 메모리상에 구축한다. 완성된 트

리는 질의 시 프리픽스가 일치하는 서브트리를 로드하여 질의를 처리할 수 있다.

 # SENoM Algorithm for Suffix Tree
Input: DNA sequence *S*, Threshold *t*
Output: Sub suffix tree *SubT*, Prefix Tree *PreT*

Algorithm Start:
Step 1. Scan *S* and partition suffixes based on first prefix character of each suffix
Step 2. Do for each partition
While (The frequency of suffixes ≤ *t*)
1. Delete *Q*
(*Q* is a data structure containing partitions)
2. Partition suffixes based on next common prefix character of each suffix
3. If the frequency of suffixes in a partition is ≤ *t*, store on the disk
Else, insert the partition into *Q*
Step 3. Make *Tsub* using suffixes in the partition
Step 4. Make *PreT* using a prefix of suffixes stored on the disk before
END

그림 1 트리구축을 위한 SENoM 인덱싱 알고리즘

예제 1. 그림 2는 전체 시퀀스 S = ACCAGCATT이고, 임계치 t = 2일 때, 서픽스 트리로 구축한 모습이다.

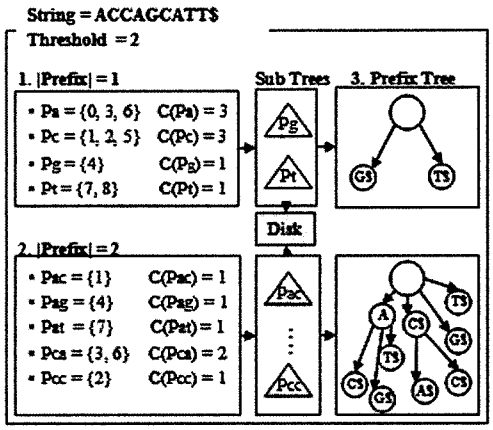

그림 2 S = ACCAGCATT 일 때, 서브 트리 생성과정

첫 번째 단계에서는, 전체 시퀀스에 대해 프리픽스의 집합 P = {*Pprefix*…}를 구성한다. 집합 P의 프리픽스의 길이는 $0 \le |prefix| < |S|$이며, 처음 S를 전체 스캔하여 $|prefix|$=1인 파티션 *Pa, Pc, Pg, Pt*를 얻는다. 여기에서, *Pa*는 *a*라는 문자를 프리픽스로 가지고 있는 서픽스들의 집합인 파티션 P를 의미한다.

파티션 *Pa* 에서의 서픽스 오프셋은 {0, 3, 6}이며, 해당 문자열 집합은 {ACCAGCATT$, AGCATT$, ATT$}이다. 파티션 *Pc*의 경우 서픽스 오프셋은 {1, 2, 5}이며, 해당 문자열은 {CCAGCATT$, CAGCATT$, CATT$}이다. $|prefix|$=1에서 서픽스들의 발생빈도수를 계산하면, $C(Pa)$=3, $C(Pc)$=3, $C(Pg)$=1, $C(Pt)$=2이므로, $C(Pprefix) \le 2$를 만족하는 파티션 *Pg*와 *Pt*는 디스크에 저장하고, 나머지는 큐에 저장한다.

두 번째 단계에서는 $C(Pprefix) > 2$인 파티션들의 분할을 위해 각 서픽스의 오프셋을 증가시킨다. *Pa*에서의 공통의 프리픽스를 제외한 서픽스의 첫번째 오프셋은{1, 4, 7}이며, 캐릭터는 {C, G, T}이다. *Pc*에서의 오프셋은 {2, 3, 6}이며, 캐릭터는 {C, A ,A}이다. 각 프리픽스의 길이가 증가하여 파티션 *Pac, Pag, Pat, Pca, Pcc*가 생성되며, 모든 파티션이 $C(Pprefix) \le 2$를 만족하므로 디스크에 저장한다.

세 번째 단계인 프리픽스 트리의 생성과정은 4장의 질의처리방안에서 자세히 다루기로 한다.

파티션 분할은 서픽스 트리 구축에서 독립적인 서브트리들을 메인 메모리에 구축함으로써 필요한 메모리 요구량을 줄인다는 장점을 가진다. 대용량의 DNA를 작은 메모리 상에서 인덱싱 할 수 있기 때문이다. 만약, N개의 스트링에서 P개의 서브트리를 구축했다면, 각 노드 당 메모리 요구량이 12byte이므로, 메모리요구량은 12 * Nbyte에서 12 * Nbyte/P로 감소한다. 그러나, 전체 시퀀스가 메인 메모리에 충분한 크기로 구축 가능하다면 파티션 분할과정은 포함하지 않아도 된다.

3.2 서픽스 노드 저장 방안

본 절에서는 구축한 서브 서픽스 트리의 노드를 저장하는 방안에 대해 TRELLIS 알고리즘[14]과의 비교, 분석을 통해 설명한다.

TRELLIS 알고리즘의 경우, Ukkonen 알고리즘[13]을 이용하여 서픽스 트리를 구축하는데, 모든 자식노드들의 정보를 저장하므로 내부노드의 경우 40byte, 리프노드인 경우 16byte의 저장공간이 요구된다. 따라서, 노드를 위한 저장공간이 많이 요구된다. 본 논문에서는 2.2절에서 설명한 Phased 알고리즘 기반의 서픽스 링크가 없는 노드구조를 사용한다.

그림 3은 트리의 노드구조를 나타낸다. 왼쪽 그림에서와 같이, 모든 노드는 문자열의 시작 오프셋, 왼쪽 자식노드에 대한 포인터, 오른쪽 형제노드에 대한 포인터 정보로 구성되며, 노드 당 메모리 요구량은 12byte이다.

오른쪽 그림은 i 번째까지 저장된 트리 배열의 구조이다. 회색으로 채워진 엔트리는 문자열에서의 시작 위치를 의미하며, 나머지 두 개의 엔트리는 형제노드와 자식노드의 인덱스 위치를 표시한다. 인덱스 위치는 서브트리가 디스크에 저장되어 다시 메모리에 할당될 때, 같

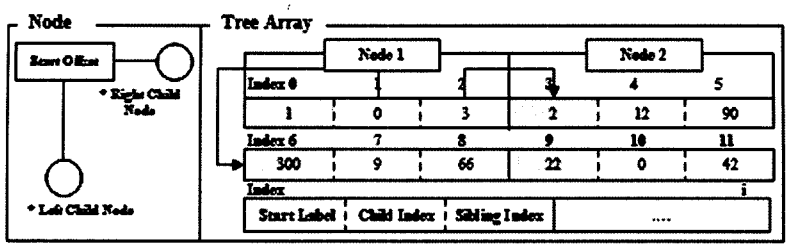

그림 3 트리노드 인덱스 정보

은 위치의 인덱스를 가리키기 위해 필요하다. 루트노드인 첫번째 노드의 경우, 2번째 인덱스의 값인 자식노드의 위치로 3을 가지므로 3번째 인덱스의 가리키며, 형제노드의 값인 1번째 인덱스 값은 형제노드가 존재하지 않으므로 0을 저장한다.

위와 같은 저장방식은, TRELLIS 알고리즘과 비교하여, 리프노드의 경우 4byte, 내부노드의 경우 28byte만큼 메모리 요구량을 줄일 수 있다.

3.3 합병하지 않는 파티션 분할방안

본 절에서는 제안하는 SENoM의 파티션 분할방안과 TRELLIS 알고리즘의 파티션 분할방식을 비교, 분석한다.

대부분의 서픽스 트리 구축 알고리즘은 공간 부하로 인해 메모리 요구량이 크며, 낮은 참조국부성으로 버퍼 관리가 어렵다는 문제점을 가진다[14]. 이러한 문제점을 해결하기 위하여, 많은 연구에서 디스크 기반의 분할방식을 사용하고 있다.

그림 4는 TRELLIS 알고리즘과 제안하는 SENoM의 파티션 분할방식에서의 차이를 나타낸다. 그림에서 사각형은 문자열 집합을, 삼각형은 서픽스 트리를 의미하며, 숫자는 각 단계를 의미한다.

그림 4(a)는 TRELLIS 알고리즘의 분할방식으로, 첫 번째 단계에서는 전체 시퀀스의 순차적 분할을 통해 파티션 *part1, part2, part3*를 생성한다. 두 번째 단계에서는 *part1, part2, part3*를 서픽스 트리로 구축하고, 이를 디스크에 저장한다. 세 번째 단계는 두번째 단계에서 구축한 서픽스 트리를 공통의 프리픽스를 가지는 서브트리 *SubT1, SubT2, SubT3*로 분할하는 과정이다. 마지막 단계에서는 서브트리를 완전한 서픽스 트리로 구축하기 위해, 공통의 프리픽스를 가지는 서브트리 간 합병한다. 그림에서 회색으로 채워진 삼각형은 공통의 프리픽스를 가지는 서브트리를 의미하므로, 해당 서브트리는 *Mi*로 합병된다.

TRELLIS 알고리즘에서는 다음과 같은 방식으로 합병이 이루어진다. 첫 번째 단계에서 *SubTi*가 *i*번째 서브트리 이고, *Mi*가 *SubTi*까지 합병된 트리라고 가정했을 때, 다음단계에서는 *i*+1번째 서브트리 *SubTi*+1를 메모리에 로드하고 합병하여 *Mi*+1을 얻는다. 모든 서브 서픽스 트리가 합병되면, 디스크에 다시 저장된다.

본 논문에서는 합병과정을 제거하기 위해, 그림 4의 (b)와 같이 Wotdeager 알고리즘 기반의 트리를 구축한다. 첫 번째 단계에서는 전체 스트링에서 공통의 프리픽스를 가지는 서픽스들간의 집합을 생성한다. 그림에서 회색부분은 공통의 프리픽스를 가지는 부분을 의미하며, 이는 *Suffix Set*으로 합쳐져 디스크에 저장된다. 이에 대한 자세한 설명은 4장에서 논하고 있다. 두 번째 단계에서는 각 *Suffix Set*을 서픽스 트리인 *Sub Tree*로 구축한다.

TRELLIS의 합병과정은 프리픽스를 공유하는 서브트리에 대해서 완성된 트리를 구축할 때마다, 최악의 경우 $O(n^2)$의 비용이 발생하게 된다. 또한, 각 서브트리의 프리픽스를 구하기 위해, DNA 시퀀스에 대한 디스크 랜덤 엑세스가 발생한다.

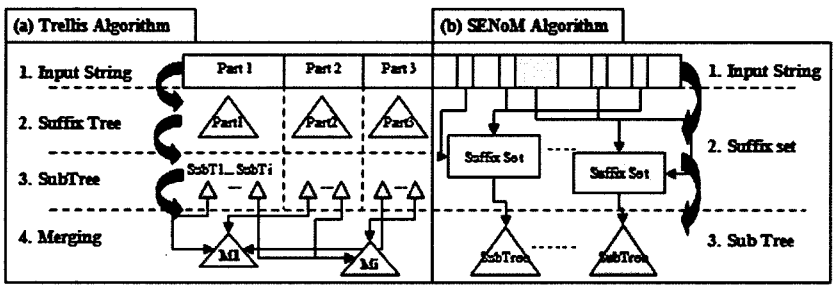

그림 4 (a) TRELLIS 파티션 분할 방안과 (b) SENoM의 파티션 분할 방안

본 논문에서는 전체 스트링을 임계치 이하의 값을 가지는 서브파티션으로 분리하지 않고, 전체 스트링에서 임계치 이하인 파티션을 서브트리로 생성한다. 이러한 방법은, 각 파티션이 모두 독립적인 프리픽스를 가지게 되므로, 파티션 간 복잡한 합병 과정을 수행하지 않아도 된다.

4. 질의 처리 방안

본 장에서는 3장에서 구축한 서픽스 트리 인덱스를 이용한 검색기법에 대해 논의한다.

DNA 시퀀스에서 근사 문자열 매칭의 경우, 일반적으로 다이나믹 프로그래밍(Dynamic Programing) 기법을 사용한다[2]. 다이나믹 프로그래밍 기법은 비교할 두 시퀀스에 대하여 2차원의 다이나믹 프로그래밍 테이블을 생성하고, 시퀀스 간 최적의 거리를 얻는 방법이다. 최적의 거리를 계산하기 위하여 일반적으로 에디트 거리(Edit Distance) 등이 유사도 함수로 사용된다[4]. 그러나, 본 절에서는 설명의 편의를 위해 하나의 문자열로 된 정확 문자열 매칭 질의를 다룬다.

본 논문에서는 질의처리를 위해 파티션 생성과정에서 모든 서브트리의 프리픽스를 서픽스로 가지는 트리를 메모리상에 구축하며, 이를 프리픽스 트리라 지칭한다.

예제 1. 그림 5는 전체 시퀀스 S = ACCAGCATT 일 때, 프리픽스 트리의 구축과정을 나타낸다. 파티션 분할과정과 서픽스 트리 구축 단계는 3.1절의 그림 2와 같다. 서픽스의 빈도발생수가 2를 초과하지 않는 파티션은 {*Pg, Pt, Pac, Pag, Pca, Pcc*}이며, 해당 파티션의 프리픽스는 *Prefix* = {G$, T$, AC$, AG$, AT$, CA$, CC$}이다. 각 파티션의 프리픽스는 그림 오른쪽에 보여지는 프리픽스 트리의 서픽스가 되어 트리로 구축된다. 트리에서 점선화살표는 각 리프노드가 루트에서부터 리프노드까지의 라벨을 프리픽스로 가지는 서브트리를 가리킨다.

질의 문자열과 정확히 일치하는 부분을 찾기 위한 방법은 그림 6의 알고리즘과 같이 두 단계로 구성된다.

Step 1. 프리픽스 트리를 로드하여, 주어진 질의 문자열을 포함하는 파티션을 탐색하는 과정이다. 탐색은 프리픽스 트리의 루트에서부터 시작하며, 질의문자열의 프리픽스가 프리픽스 트리의 리프노드까지 모두 일치하는 경우, 해당 문자열을 프리픽스로 가지는 서브트리를 디스크에서 찾는다.

Step 2. Step 1에서 찾은 파티션에서, 나머지 질의 문자열을 포함하는지 여부를 검사하는 과정이다. 구축한 서픽스 트리는 라벨 시작 위치와 왼쪽 자식 노드, 오른쪽 형제노드만 저장할 뿐, 모든 형제노드나 자식노드, 라벨의 마지막 위치에 대한 정보를 별도로 저장하지 않

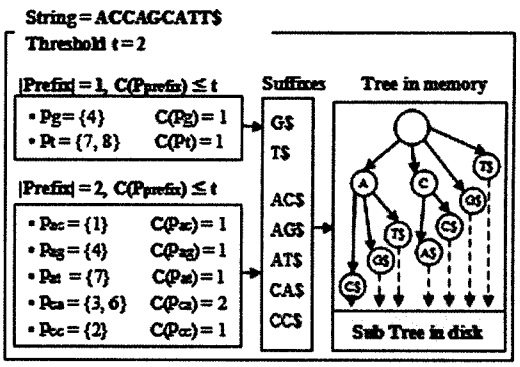

그림 5 프리픽스 트리 생성

#Algorithm for Query
Input: DNA sequence *S*, Query *q*, Prefix tree file *Pfile*
Output: Matched Position *mp*

Algorithm Start:
Step 1. Load *Pfile*
While (Trace nodes in the Prefix tree)
Compare label in Prefix tree with *q*
If(Not matched with part of *q*) exit
If(Leaf node) break
Return *mp*
Step 2. Load a subtree which prefix is the matched character on step1
While (Trace nodes in the subtree)
Compare label in subtree with *q*
If(Not matched with *q*) exit
If(End of tree or *q*) Return a *mp*
END

그림 6 질의처리 알고리즘

는다. 따라서 서픽스 트리를 탐색할 때, 노드의 자식노드로부터 라벨 마지막 위치 정보를 얻고, 형제노드 또는 자식노드로의 탐색과정이 필요하다.

본 논문에서 제안하는 질의 처리 방안은, 질의할 때마다 모든 파티션에 대한 정보를 파일에서 순차적으로 읽지 않아도 된다는 장점을 가진다. 파티션의 수가 매우 많은 경우에도, 질의와 일치하는 서브트리를 찾기 위해 프리픽스 트리를 탐색하여 빠른 시간 내에 질의처리가 가능하다.

5. 실험 및 분석

본 장에서는 실험에 의한 성능 분석을 통하여, 제안하고 있는 SENoM의 우수성을 증명한다. 5.1절에서는 SENoM의 복잡도를 분석하고, 5.2절에서는 실험환경에 대해 설명한다. 5.3절에서는 TRELLIS 알고리즘과 SENoM을 비교, 분석하여 실험의 결과를 보인다.

5.1 합병하지 않는 파티션 분할방안

본 절에서는 실험에 앞서, SENoM의 시간 복잡도와 디스크 I/O를 분석, 비교한다. SENoM은 파티션 생성 단계, 서브트리 생성 단계로 구성된다.

먼저, 파티션 생성단계에서 공통의 프리픽스를 가지는 서픽스들의 집합으로 분리되는데, 최대 t개의 서픽스들을 가지는 파티션으로 생성된다. 각 파티션의 구축을 위한 비용이 $O(p)$이고, 파티션의 수가 전체 데이타을 임계치로 나눈 값인 Sizeof(*Input*)/t개일 때, 생성 시간은 $O(p)*$ (Sizeof(*Input*)/t) = $O(n)$이다.

두 번째로, 트리 생성단계에서는 구축되는 서브트리의 개수는 파티션의 개수와 동일하고, 각 파티션의 길이는 최대 t이므로, 트리 구축시간은 Sizeof(*Input*)/t*$O(t)$이 된다. 또한, 모든 서브트리가 디스크에 저장되므로, O(Sizeof(*Input*)/t)의 디스크 엑세스가 요구된다.

TRELLIS 알고리즘에서는 합병단계가 있어, 서브트리 간 LCP를 찾을 때까지 노드의 자식 노드를 따라가며, 합병과정을 수행한다. 합병을 위한 수행시간이 p이고, 각 노드에서 4개의 자식노드를 위한 에지를 탐색하므로 합병을 위한 탐색시간은 $4p$이다. 그러나 최악의 경우, LCA를 찾기 위해 전체 트리를 순회해야 하며, 전체 트리의 에지와 트리노드의 수가 $O(n)$이므로, $O(n^2)$의 비용이 발생하게 된다. SENoM에서는 합병단계가 생략되어 합병을 위한 시간 탐색비용이 필요하지 않다.

5.2 실험 환경

본 실험의 목적은 SENoM의 성능이 기존의 알고리즘에 비해 우수함을 보이는 것이다. 실험 데이터는 난수 발생기로부터 DNA 비율(A 30%, C 20%, G 20%, T 30%)에 따라 랜덤 추출한 DNA 시퀀스를 사용하며, 임계치 계산을 위한 메모리 가용량은 10000MB일 때는 600MB, 나머지의 경우, 512MB로 설정한다. 그 이유는, 임계치는 실험 데이터의 크기에 따라 비례적으로 감소하므로, 실험데이터가 매우 큰 경우 임계치에 따라 파티션의 분할이 증가하여 서브트리의 크기가 불필요하게 작아질 수 있기 때문이다.

실험은 1GB 메모리를 가진 AMD Athlon 64 3700+ (2.2GHz) PC에서 동작하며, 플랫폼으로 Debian Linux를 사용한다. TRELLIS 알고리즘은 저자로부터 받은 C++ 코드이며, 제안하는 SENoM 알고리즘 또한 C++ 로 구현하여, GNU g++ 컴파일러 버전 4.1.2에서 컴파일 하였다.

5.3 실험 결과

본 절에서는 SENoM의 성능을 평가하기 위하여, 버퍼 크기, 메모리 요구량, 트리 구축시간, 질의처리 시간, 인덱스 크기 등을 TRELLIS 알고리즘과 비교하여 측정한다.

5.3.1 버퍼 크기

표 2는 버퍼 크기의 변화에 따른 트리 구축시간을 측정한 결과이다. 실험은 DNA 시퀀스의 크기와 버퍼의 크기를 로그 스케일로 증가시키면서 실험하였다. 실험결과를 통해, 버퍼의 크기가 1MB 일 때, 트리 구축 성능이 최적화됨을 알 수 있다. 따라서, 본 논문에서의 버퍼 크기는 1MB로 고정하여 실험한다. TRELLIS 알고리즘에서도 최적화된 버퍼의 크기로 1MB를 고정하여 실험하였다.

표 2 버퍼크기에 증가에 따른 트리 구축 시간(secs) 비교

Buffer/Input	1MB	10MB	100MB
0.1 Buffer	1.74	25.18	365.43
1 Buffer	1.68	25.06	363.31
10 Buffer	1.75	25. 65	366.77
100 Buffer	2.06	25.70	364.16

5.3.2 트리 구축 시간

그림 7은 실험 데이터 크기의 증가에 따른 트리 구축 시간을 TRELLIS 알고리즘과 비교하여 측정한 결과이다. DNA 시퀀스의 크기는 200MB, 400MB, 600MB, 800MB, 1000MB로 증가시키면서, 전체 트리 구축 시간을 측정하였다. 트리 구축시간은, 프리픽스의 생성과 파티션 분할, 합병과 서브트리 생성단계를 모두 포함한 시간이며, 공정한 실험을 위해, TRELLIS에서 서픽스 링크구축을 위한 시간은 제외한다. 또한, 실험 데이터의 패턴에 따라 구축시간의 차이가 발생하므로, 이를 평균하여 결과에 나타내었다.

실험결과를 통해, SENoM과 TRELLIS 알고리즘의 구축속도가 거의 동등함을 알 수 있다. 본 논문에서 구축속도가 거의 동등한 이유는, 파티션을 생성하기 위해 전체 DNA 시퀀스를 스캔하는 작업을 반복하여 수행하기 때문이다. 그러나, TRELLIS 알고리즘은 트리구축과

Input(MB)	SENoM(secs)	TRELLIS(secs)
200	852.42	815.62
400	1748.84	1671.84
600	2550.06	2546.67
800	3533.56	3540.78
1000	4335.64	4528.16

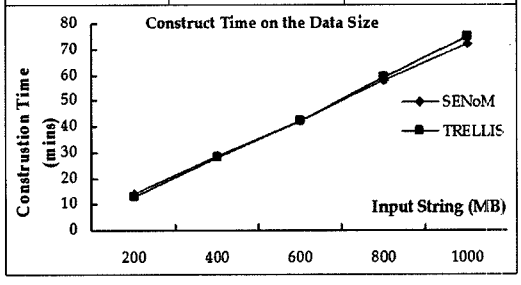

그림 7 데이터 크기의 증가에 따른 트리 구축 시간 비교

Input(MB)	SENoM(secs)	TRELLIS(secs)
200	148	266
400	335	550
600	556	864
800	790	1139
1000	996	1496

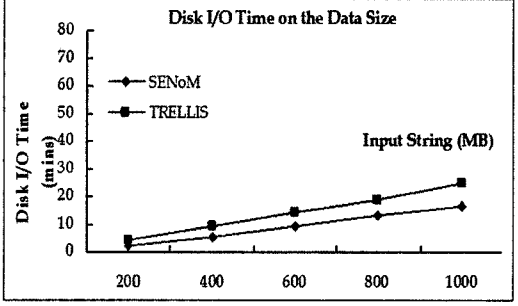

그림 8 데이터 크기 증가에 따른 디스크 I/O 시간 비교

Input(MB)	SENoM(MB)	TRELLIS(MB)
200	512	775
400	518	753
600	502	721
800	506	690
1000	512	740
12000	512	790

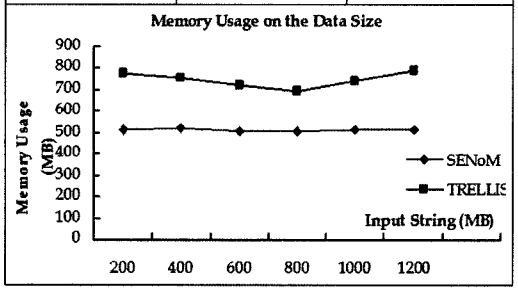

그림 10 데이터 크기 증가에 따른 메모리 요구량 비교

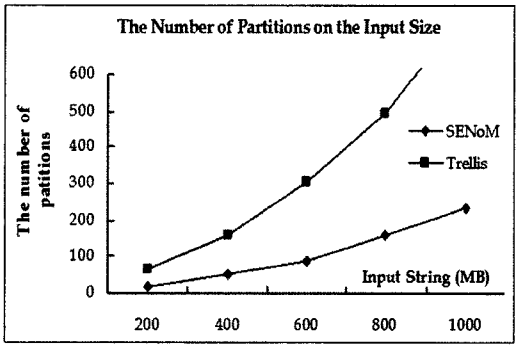

그림 9 데이터 크기 증가에 따른 파티션 수 비교

정에서 그림 8과 같이 디스크 I/O를 위한 처리시간이 많이 발생한다.

TRELLIS 알고리즘에서는 전체 시퀀스를 서브 시퀀스로 분할하고, 각 서브시퀀스에 대해 서브 파티션을 구성한다. 따라서, 그림 9와 같이 전체 시퀀스가 1000MB일 때에는 본 논문은 232개의 파티션, TRELLIS는 783개의 파티션이 발생한다. 파티션이 많이 발생하는 경우 합병비용이 커지며, 해당 파티션으로 접근하기 위한 디스크 I/O 오버헤드가 발생한다.

5.3.3 메모리 요구량

그림 10은 실험데이터 크기의 증가에 따른 메모리 요구량을 TRELLIS 알고리즘과 비교하여 측정한 결과이다. DNA 시퀀스의 크기는 200MB, 400MB, 600MB, 800MB, 1000MB, 1200MB로 증가시키면서, 트리 구축에 필요한 메모리 요구량을 측정하였다. 실험결과를 통해, SENoM의 경우 평균 510MB의 메모리, TRELLIS의 경우 750MB의 메모리를 사용하여, 본 논문의 메모리 요구량이 적음을 알 수 있다. 2장에서 언급한 TDD 알고리즘의 경우, 본 논문과 비교하여 더 적은 공간을 요구하지만, 쏠림현상의 발생과 트리구축시간이 2배 이상 소요된다는 문제점을 가진다.

TRELLIS 알고리즘은 내부노드를 위한 메모리로 40byte, 리프노드를 위한 메모리로 16byte를 할당받기 때문에, 내부노드와 리프노드 각각 12byte를 할당 받는 본 연구와 비교하여 메모리 요구량이 크다. 따라서 전체 트리 구축을 위해 요구되는 메모리 량 역시 크다.

그림 10에서, TRELLIS 알고리즘은 데이터의 크기가 커질수록 메모리 요구량이 줄어드는 구간이 존재하는데, 이는 데이터 증가율보다 임계치의 감소율이 근소하게 크기 때문이다.

5.3.4 디스크 요구량

그림 11은 실험데이터 크기의 증가에 따른 디스크 요구량을 TRELLIS 알고리즘과 비교하여 측정한 결과이다. DNA 시퀀스의 크기는 200MB, 400MB, 600MB, 800MB, 1000MB로 증가시키면서, 트리 구축에 필요한 디스크 요구량을 측정하였다. 실험결과를 통해, 두 알고리즘 모두 데이터의 크기변화에 비례하여 인덱스 크기가 증가함을 알 수 있다. 그러나, 절대 크기를 비교하는 경우, SENoM이 TRELLIS 알고리즘보다 인덱스를 위한 디스크 요구량이 적음을 알 수 있다.

TRELLIS 알고리즘에서는, 내부노드의 경우 시작 위치, 마지막 위치, 모든 자식노드에 대한 정보를 위해 총 28byte를 저장하며, 리프노드의 경우 시작 위치와 서픽스 위치 등 총 8byte를 저장한다. 본 논문에서는 내부노드와 리프노드 모두 12byte를 저장하게 되므로 TRELLIS의 약 80%정도의 저장공간만을 필요로 한다.

Input(MB)	SENoM(GB)	TRELLIS(GB)
200	4.1	5.0
400	8.2	10.1
600	12.3	15.1
800	16.4	20.2
1000	20.5	25.2

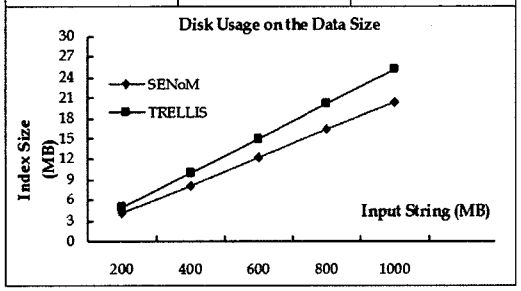

그림 11 데이터 크기의 증가에 따른 인덱스 크기 비교

5.3.5 질의처리 시간

그림 12는 질의데이터 크기의 증가에 따른 측정한 결과이다. 질의 데이터는 로그스케일로 증가시키면서 실험 데이터에서 랜덤하게 추출하여 실험하였다. 각 실험은 100번의 반복 실험 결과를 평균하여 나타낸다.

질의 처리 시간은, 질의를 만족하는 서브시퀀스를 검색하여, 일치하는 오프셋을 반환하는 총 시간을 의미한다. 실험결과를 통해, TRELLIS 알고리즘의 질의처리 속도가 SENoM의 질의처리 속도와 비교하여, 근소하게 빠른 것을 알 수 있다. 그 이유는, TRELLIS 알고리즘은 질의 처리시 인덱스 정보로 저장된 라벨 시작위치와 마지막 위치로 노드의 길이를 계산하는 반면, SENoM

Query(bp)	TRELLIS(secs)	SENoM(secs)
100	0.0333	0.0410
200	0.0283	0.0379
400	0.0331	0.0391
800	0.0354	0.0410
1600	0.0337	0.0497
3200	0.0376	0.0430
6400	0.0297	0.0354
12800	0.0364	0.0429

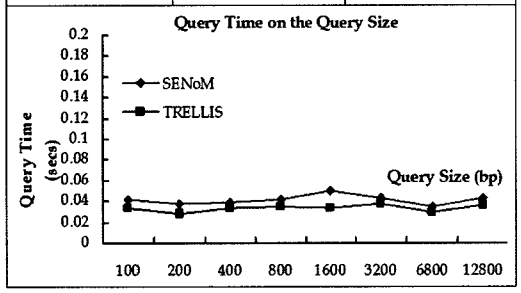

그림 12 질의 데이터 크기 증가에 따른 질의 처리 시간

에서는 노드의 길이를 계산하기 위해 자식노드에 접근하기 때문이다.

그러나, TRELLIS 알고리즘과 SENoM의 속도차이는 평균 0.0066초로 매우 근소한 차이이며, 질의 데이터의 크기와 상관없이 0.05초 이내에 질의처리가 가능하다.

6. 결 론

본 논문에서는 효과적인 DNA 서열의 유사성 검색을 위해, 디스크 기반의 서픽스 트리 인덱싱 방안인 SENoM을 제안하였다. SENoM은 가변길이의 프리픽스 방식으로 쏠림현상을 해결하고, 전체 구조를 트리로 구축하여 복잡한 합병과정을 제거하는 공간 효율적인 인덱싱 방안이다. SENoM은 전체 시퀀스를 스캔하면서, 공통의 프리픽스를 찾고, 공통의 프리픽스를 가지는 서픽스들의 발생 빈도수가 임계치(Thredhold) 이상인 경우, 프리픽스를 확장하여 파티션을 진행한다. 서픽스들의 발생 빈도수가 임계치(Thredhold) 이하인 경우, 해당 파티션의 각 서픽스를 서브트리로 구축한 후, 디스크에 저장하였다. 다음으로 서브트리로 구축된 파티션의 프리픽스를 서픽스로 가지는 트리를 메모리상에 구축하였다.

SENoM의 우수성을 검증하기 위해, 실험을 통한 성능 평가를 실시하였다. 실험결과에 의하면, SENoM은 2GB의 메모리 상에서 TRELLIS 알고리즘에 비해, 약 35% 낮은 메모리 할당량으로 1G의 DNA 시퀀스를 2GB의 메모리에 71분 내에 인덱싱 할 수 있는 것으로 나타났다. 또한 SENoM은 TRELLIS 알고리즘의 약 20% 낮은 인덱스 크기를 가지면서, 동시에 질의가 긴 경우에도, 0.05초 내에 효율적으로 검색하는 것으로 나타났다.

향후, 합병하지 않는 디스크 기반의 서픽스 트리 인덱싱 기술을 사용함으로써 빠른 검색속도를 제공할 수 있는 DNA 시퀀스 검색 시스템 구현에 활용될 수 있을 것으로 판단된다.

참 고 문 헌

[1] S. F. Altschul et al, "Basic Local Alignment Search Tool," *Journal of the Molecular Biology*, vol.215, no.3, pp.403-410, 1990.

[2] E. McCreight, "A Space-Economical Suffix Tree Construction Algorithm," *Journal of the ACM*, vol.15, no.2, pp.514-534, 1976.

[3] D. G. Jeffrey et al, "BeoBLAST: Distributed BLAST and PSI-BLAST on a Beowulf cluster," *Journal of the Bioinformatics*, vol.18, no.5, pp.765-766, 2002.

[4] E. Hunt, M. P. Atkinson and R. W. Irving, "Database Indexing for Large DNA and Protein Sequence Collections," *Journal of the VLDB*, vol.11, no.3, pp.256-271, 2002.

[5] D. Yao, C. Shahabi and P. Å. Larson, "Hash-based Labeling Techniques for Storage Scaling," *Journal of the VLDB*, vol.14, no.2, pp.222-237, 2005.
[6] P. Krishnan, J. S. Vitter and B. R. Iyer, "Estimating Alphanumeric Selectivity in the Presence," In *Proc. of the 1996 ACM SIGMOD Int'l Conf. on Management of Data*, vol.25, no.2, pp.282-293, June, 1996.
[7] M.Farach-Colton, P.Ferragina and S.Muthukrishnan, "Overcoming the Memory Bottleneck in Suffix Tree Construction," *Journal of the ACM*, vol.47, no.6, pp.987-1011, 2007.
[8] C. F. Cheung, J. X. Yu and H. Lu, "A Compact Partitioned Suffix Tree for Disk-based Indexing on Large Genome Sequences," *IEEE Trans. on Knowledge and Data Engineering*, vol.17, no.1, pp.90-105, 2005.
[9] Y. Tian et al., "Practical Methods for Constructing Suffix Trees," *Journal of the VLDB*, vol.14, no.3, pp.281-299, 2005.
[10] S. Tata, R. Hankins and J. Patel, "Practical Suffix Tree Construction," In *Proc. of the VLDB Int'l Conf.*, vol.23, no.2, pp.36-47, 2004.
[11] J. I. Won, S. K. Hong, J. H. Yoon, S. H. Park and S. W. Kim, "A Practical Method for Approximate Subsequence Search in DNA Databases," In *Proc. of the PAKDD'2007*, pp.921-931, 2007.
[12] R. Giegerich, S. Kurtz and J. Stoye, "Efficient Implementation of Lazy Suffix Trees," In *Proc. of the Workshop on Algorithm Engineering*, vol.33, no.11, pp.1035-1049, 2003.
[13] E. Ukkonen and J. Karkkainen, "On-line Construction of Suffix Trees," *Journal of the Association for Computing Machinery*, vol.14, no.3, pp.262-272, 1995.
[14] B. Phoophakdee and M. J. Zaki, "Genome-scale Disk-based Suffix Tree Indexing," In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pp.833-844, 2007.
[15] E. Hunt, M. Atkinson and R. Irving, "A Database Index to Large Biological Sequences," In *Proc. of the VLDB Int'l Conf.*, vol.7, no.3, pp.139-148, 2001.
[16] S .Bedathur and J. Haritsa, "Engineering a Fast Online Persistent Suffix Tree Construction," In *Proc. of the IEEE 20th Int'l Conf. on Data Engineering*, vol.20, pp.720-731, 2004.

송 혜 주

2003년 3월~2007년 2월 숙명여자대학교 이과대학 멀티미디어과학과(학사). 2007년 3월~2009년 2월 숙명여자대학교 일반대학원 멀티미디어과학과(이학석사). 2009년 6월~현재 롯데정보통신 연구원재직 관심분야는 데이터베이스, 멀티미디어데이터베이스, Bio정보공학

박 영 호

1986년 3월~1992년 2월 동국대학교 공과대학 컴퓨터공학과(학사, 공학석사). 1993년 8월~1999년 2월 한국전자통신연구원(ETRI) 교환전송연구단 선임연구원 1999년 3월~2005년 8월 한국과학기술원(KAIST) 전산학과(공학박사, DBMS) 2005년 9월~2006년 2월 한국과학기술원(KAIST) 첨단정보기술연구센터 연구원. 2001년 9월~2006년 2월 한국산업기술대학교(KPU) 컴퓨터공학과 겸임교수. 2005년 9월~2006년 2월 동국대학교 컴퓨터멀티미디어학과 겸임교수. 2006년 3월~현재 숙명여자대학교 이과대학 멀티미디어과학과 조교수. 관심분야는 데이터베이스, XML, IR(정보검색), 멀티미디어데이터베이스, Bio정보공학, 영상미디어, 예술&공학인터페이스

노 웅 기

1991년 2월 한국과학기술원(KAIST) 전산학과 학사. 1993년 2월 한국과학기술원(KAIST) 전산학과 석사(멀티미디어 전공). 2001년 2월 한국과학기술원(KAIST) 전산학과 박사 (데이터 마이닝 전공). 2001년 2월~2003년 9월 ㈜티맥스소프트 책임연구원(미들웨어 개발). 2003년 10월~2005년 3월 ㈜티맥스데이타 수석연구원(DBMS 엔진 개발). 2005년 4월~2006년 5월 한국과학기술원 전산학과 초빙교수. 2006년 6월~2007년 7월 Visiting Scholar, University of Minnesota, USA. 2007년 8월~2008년 2월 NHN㈜ 수석연구원(대용량 로그 데이터 분석). 2008년 3월~현재 성결대학교 멀티미디어학부 전임강사. 관심분야는 대용량 데이터 마이닝, 데이터웨어하우징, 정보 검색, 멀티미디어 데이터베이스, 멀티미디어 정보 검색, 데이터베이스 시스템 엔진