DNA 시퀀스 검색을 위한 효율적인 인덱스 기법

홍상균°*, 원정임**, 윤지희*
*한림대학교 정보통신공학부, *한양대학교 정보통신학부
kyoons@hallym.ac.kr°, jiwon@hanyang.ac.kr, jhyoon@hallym.ac.kr

An Efficient Index Structure for DNA Sequence Retrieval

Sang-Kyoon, Hong°*, JungIm Won**, JeeHee Yoon*
*Division of Information and Communication Engineering, Hallym University
**Division of Information and Communications, Hanyang University

요 약

DNA 시퀀스 데이터베이스 규모의 급격한 증가 추세를 고려할 때, DNA 시퀀스 검색 연산을 보다 효과적으로 지원할 수 있는 인덱싱 및 질의 처리 기술이 요구 된다. 접미어 트리는 DNA 시퀀스 검색을 위한 좋은 인덱스 구조로 알려져 왔다. 그러나 접미어 트리는 그 구조적 특성으로 인하여 저장공간, 검색 성능, DBMS와의 통합 등의 문제점을 갖는다. 본 논문에서는 이와 같은 접미어 트리의 문제점들을 해결하는 DNA 시퀀스 검색을 위한 새로운 인덱스 구조를 제안하고, 이를 기반으로 하는 효율적인 질의 처리 방식을 제안한다. 제안된 인덱스 기법은 이진 트라이를 기본 구조로 채택하며 DNA 시퀀스의 윈도우 서브 시퀀스를 인덱싱 대상으로 한다. 유사 서브 시퀀스 검색을 위한 질의 처리 알고리즘은 기본적으로 다이나믹 프로그래밍 기법에 근거하여 이진 트라이를 루트로부터 너비 우선(breadth-first) 방식으로 운행하며, 경로 상에 존재하는 모든 유사 서브 시퀀스를 검색해 낸다. 제안된 기법의 우수성을 검증하기 위하여, 기존의 접미어 트리와의 비교 실험을 통한 성능 평가를 수행하였다. 실험 결과에 의하면, 제안된 인덱스 기법은 접미어 트리에 비하여 약 30%의 작은 저장 공간을 가지고도 수배에서 수십배의 검색 성능의 개선 효과를 나타낸다.

1. 서론

모든 생물은 각각의 특성을 결정하는 A, C, G, T의 네 가지 문자들로 구성된 DNA 시퀀스 코드를 가지고 있다. 분자 생물학에서 임의의 DNA 시퀀스의 의미를 해석하기 위한 기본적인 방법은 그것과 염기 배열이 유사한 다른 DNA 시퀀스를 찾아 그 시퀀스의 특징을 이용해 기능을 유추하는 것이다[1].

DNA 시퀀스 검색을 위한 일반적인 방법은 데이터베이스 전체를 검색하는 순차 검색 기법이다[2]. DNA 시퀀스 검색 연산을 위하여 가장 널리 사용되는 표준 도구에는 BLAST[3,4]가 있다. BLAST는 고속의 시퀀스 검색 기능을 제공하지만, 검색 결과의 정확도가 완전하지 못하다. 정확성을 보장할 수 있는 대표적인 알고리즘에는 Smith-Waterman 알고리즘[2]을 들 수 있다. 그러나, 이 방식은 계산량이 많아 속도가 느리므로 실제적인 사용에는 제한적이다.

최근의 DNA 시퀀스 데이터베이스의 규모의 급격한 증가 추세를 고려할 때, DNA 시퀀스 검색 연산을 보다 효과적으로 지원할 수 있는 인덱싱 및 질의 처리 기술이 요구 된다. 접미어 트리(suffix tree)[5]는 DNA 시퀀스 검색을 위한 좋은 인덱스 구조로 알려져 왔다. 접미어 트리는 주어진 시퀀스의 모든 접미어 서브 시퀀스들을 트리 형태로 저장한 것으로 질의와 정확히 일치하는 서브 시퀀스를 고속으로 검색한다. 또한 DNA 시퀀스를 대상으로 하는 효율적인 유사 서브 시퀀스 검색 알고리즘들이 제안되어 있다[6-9]. 그러나 이 들 접미어 트리 기반의 유사 검색 알고리즘은 질의 시퀀스의 길이 혹은 유사 허용치 T의 크기가 커지는 경우, 인덱스의 검색 공간이 증가하여 성능이 저하하는 단점을 갖는다. 그 예로서 다음의 그림 1에 질의 시퀀스의 길이 혹은 유사 허용치 T의 크기 변화에 따른 접미어 트리 기반의 유사 서브 시퀀스 알고리즘의 검색 성능을 비교한 결과를 보인다. 실험 데이터로는 28.6M의 21번 human chromosome DNA 시퀀스를 사용하였으며, 실험 결과로부터 질의 길이 증가와 유사 허용치 증가에 따라 질의 처리 시간이 급격히 증가하고 있음을 보인다. 이와 같은 단점을 보완하기 위한 방법으로서, 참고 문헌 [10]에서는 주어진 질의 시퀀스를 작은 길이의 서브 질의로 분해하여 각 서브 질의에 대하여 작은 T 값을 대상으로 유사 검색을 수행한 후, 그 결과를 결합하는 하이브리드 인덱싱 기법을 제안하고 있다.

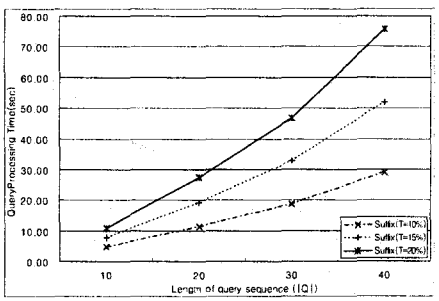

그림 1. 질의 길이와 유사도에 따른 질의 처리 시간

그러나 접미어 트리는 그 구조적 특성으로 인하여 여전히 다음과 같은 문제점을 갖는다.

① 저장 공간: 접미어 트리는 일반적으로 매우 큰 저장 공간을 필요로 하여, 그 크기는 데이터베이스 크기의 수 십배에 이른다[10-12].

② 검색 성능: 디스크 액세스 패턴이 지역적이지 못하여 검색 성능이 떨어진다. 접미어 트리의 크기가 클수록 이 트리를 순회하는 시간이 길어지며, 따라서 전체 검색 성능이 떨어진다[13].

③ DBMS와의 통합: 접미어 트리는 그 구조적 특성 상 페이지 단위로 구현하기가 어렵다. 따라서 모든 데이터를 페이지 단위로 구현하는 DBMS와 밀 결합(seamless integration)에 어려움이 있다[5,14].

본 논문에서는 이와 같은 접미어 트리의 문제점들을 해결하는 DNA 시퀀스 검색을 위한 새로운 인덱스 구조를 제안하고 이를 기반으로 하는 효율적인 질의 처리 방식을 제안한다. 제안하는 인덱스는 이진 트라이를 기본 구조로 사용하여 포인터 없이 트라이를 비트 스트링으로 표현하는 방식을 채택하며, DNA 시퀀스를 구성하는 모든 문자의 시작 위치로부터 일정 길이의

윈도우 서브 시퀀스를 추출하여 이를 인덱싱 대상으로 한다. 제안된 인덱스 기반의 유사 검색 알고리즘은 기본적으로 다이나믹 프로그래밍 기법을 사용하여 이진 트라이의 경로 상에 존재하는 모든 유사 서브 시퀀스를 검색한다. 트라이 탐색 방식은 너비 우선 방식을 사용하여 인덱스의 각 페이지에 한 번의 디스크 액세스와 한 번의 순차적 처리에 의하여 모든 유사 서브 시퀀스를 검색한다. 이 때 윈도우 사이즈보다 길이가 긴 질의 시퀀스에 대해서는 탐색된 결과에 대하여 후처리 과정을 통해 유사한 모든 서브 시퀀스를 검색해낸다.

다양한 실험을 통하여 제안된 인덱스 기법과 유사 검색 알고리즘의 성능을 기존의 방식과 비교하여 정량적으로 검증한다. 실험 결과에 의하면, 제안된 기법은 기존의 접미어 트리 기반의 유사 검색 방식과 비교하여 더 작은 저장 공간을 가지고도 수 배에서 수십배까지의 검색 성능의 개선 효과를 가지는 것으로 나타났다.

2. 관련 연구

본 장에서는 DNA 시퀀스 검색에 대한 기존의 관련 연구에 대하여 요약한다.

순차적 검색을 기반으로 하는 대표적인 유사 서브 시퀀스 검색 기법으로 Smith-Waterman(SW) 알고리즘[2]을 들 수 있다. S-W 알고리즘은 다이나믹 프로그래밍(dynamic programming) 기법에 의하여 두 시퀀스 S와 Q 사이에 최대 가능 스코어를 갖는 최적의 부분 정렬(optimal local alignment)을 찾는다. 그러나 이 방식은 두 시퀀스의 길이의 곱에 비례하는 계산량($O(|Q|\times|S|)$)를 필요로 하여 속도가 느린 단점이 있다.

BLAST[3,4]는 현재 DNA 시퀀스 검색 연산을 위하여 가장 널리 사용되는 표준 도구로서 빠른 시간 내에 최적에 가까운(near optimal) 정렬을 얻는 효율적인 알고리즘으로 인정받고 있다. 그러나 BLAST는 관련 데이터베이스가 주기억 장치에 적재되어야 하며, 검색 속도가 데이터베이스 사이즈에 비례하고, 워드의 고정 길이에 따라 검색 결과의 정확도가 영향을 받는 점 등이 문제점으로 지적된다.

DNA 시퀀스 데이터베이스에 대하여 사전에 인덱스를 구성, 활용함으로써 DNA 시퀀스 검색의 속도 향상을 기대할 수 있다. 인덱스 기반의 DNA 시퀀스 검색 기법은 역 인덱스(inverted index) 방식[15-17], 다차원 인덱스 방식[18], 영속 트리 방식[9,13,19] 등으로 분류할 수 있다.

정보 검색 분야에 활용되는 역 인덱스(inverted index)를 기반으로 하는 인덱싱 기법으로 참고 문헌 [15-17]의 방식을 들 수 있다. 시퀀스 데이터베이스 내에서 일정 길이의 인터벌(interval)을 오버랩핑 시켜가며 추출하여 이 들을 워드로 하여 각 워드의 포스팅 리스트(출현 시퀀스 번호, 오프셋 정보 등 포함)를 구성하는 방식으로 속도 향상 효과를 얻을 수 있다. 참고 문헌 [15]에서는 특히 압축 인덱스 구조를 사용하여 인덱스 파일이 과다하게 커지는 단점을 보완하고 있으나, 검색 결과의 정확도가 떨어지는 점 등이 단점으로 지적되고 있다.

참고 문헌 [18]에서는 웨이블렛(wavelet) 변환에 의하여 시퀀스 데이터베이스 내의 각 서브 시퀀스를 다차원의 정수 공간으로 매핑한 후, 이 들을 다차원 인덱스 구조로 표현하고, 영역 질의, 최근접 질의를 수행하는 새로운 방식을 제안하고 있다. 제안된 방식은 작은 사이즈의 인덱스 구조를 이용하여 질의 처리 시 데이터베이스의 검색 공간을 축소 시킬 수 있는 효율적인 방식으로 간주될 수 있으나, 유사도를 계산하기 위하여 에디트 거리(edit distance) 외의 일반적인 유사도 함수를 도입하기 어려운 점 등이 단점으로 지적되고 있으며, global alignment에 한정된 검색 기법이다.

영속 트리 기반의 대표적인 인덱싱 방식으로써 접미어 트리(suffix tree)[5]를 들 수 있다. 기존에는 디스크 상에 주기억장치 용량 이상인 대규모의 접미어 트리를 구성하는데 어려움이 있었으나, 참고 문헌 [9]에서 디스크 분할 저장 방식(partitioned suffix tree)에 의하여 이 문제를 해결한 바 있으며, 참고 문헌 [20]에서는 Top-Down Disk-based에 기반한 실질적인 접미어 트리 구성 방식을 제안하고 있다. 그러나 접미어 트리 인덱스는 그 구조적 특성으로 인하여 저장 공간의 오버헤드가 매우 크다는 점, 트리를 순회하는 시간이 길다는 점, 구조적 특성 상 페이지 단위로 구현하기가 어려워 DBMS와 밀 결합이 어렵다는 점 등의 문제점을 여전히 가지고 있다.

3. 인덱싱 방안

3.1 이진 트라이

트라이[5,21]는 노드가 정보를 가지지 않고 에지에 데이터가 저장되는 트리 구조로서, 각 에지는 하나의 문자를 가지며 루트로부터 단말까지의 경로가 인덱싱의 대상이 되는 하나의 키에 해당된다.

트라이를 구현하는 효율적인 방안으로서 포인터 없는 이진 트라이(pointerless binary trie)[22]를 생각해 볼 수 있다. 포인터 없는 이진 트라이는 알파벳 Σ를 {0,1}로 제한하여 각 노드가 최대 2개의 에지를 가지도록 하며, 0의 값을 가지는 에지는 노드의 왼편에, 1의 값을 가지는 에지는 노드의 오른편에 연결하는 규칙을 적용하여 에지 정보를 생략 표현한다. 즉, 노드 당 두 비트를 할당하여 그 값이 '10'이면 노드에 왼쪽 에지만이 연결된 형태를 표현하고, '01'은 노드에 오른쪽 에지만이 연결된 형태를 표현하고, '11'은 노드에 왼쪽 에지와 오른쪽 에지가 모두 연결된 형태를 표현하고, '00'은 에지가 연결되지 않은 단말 노드의 형태를 표현한다.

본 연구에서는 포인터를 사용하지 않는 이진 트라이의 기본 개념을 활용하여 DNA 시퀀스 검색을 지원하는 인덱싱 구조를 제안한다.

3.2 윈도우 서브 시퀀스

DNA 시퀀스 검색을 위한 인덱스 구조로서 접미어 트리가 잘 알려져 있다. 접미어 트리는 주어진 시퀀스의 접미어에 해당되는 서브 시퀀스들을 트리 형태로 구성한 것으로서 이 들 접미어들이 많은 공통 접두어를 가질 때 좋은 압축 효과를 갖는다. DNA 시퀀스로부터 추출된 접미어는 소수의 문자 집합으로 생성되는 특성에 의하여 많은 공통 접두어를 가질 확률이 높다. 그러나 DNA 시퀀스로부터 추출된 이 들 접미어들은 최대 공통 접두어(LCP : Longest Common Prefix)의 크기가 일반적으로 매우 작은 특성을 갖는다. 그림 2에 human chromosome 21번 데이터의 일부인 28.6M의 DNA시퀀스에 대한 접미어들의 최대 공통 접두어 LCP의 크기 분포의 누적율을 보인다. 이 그림으로부터 LCP의 크기가 15까지 이르러 이미 누적율이 약 83%에 달하며, LCP의 크기가 20까지의 경우에는 90%이상의 누적율을 보임을 알 수 있다.

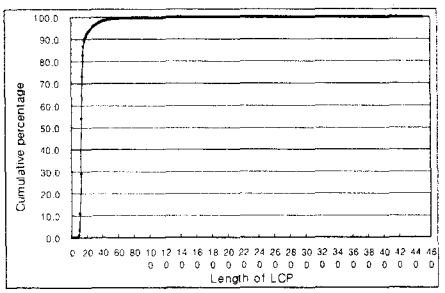

그림 2. LCP 누적 분포도

이와 같은 DNA 시퀀스의 특성을 고려하여 본 연구에서는 DNA시퀀스의 접미어 전체를 인덱싱 대상으로 하지 않고, 일정 길이의 접두어 서브 시퀀스만을 인덱싱 대상으로 한다. 즉 DNA 시퀀스의 가능한 모든 위치에 일정 길이 |W|의 슬라이딩 윈도우를 위치시켜, 이 슬라이딩 윈도우에 덮여지는 서브 시퀀스를 인덱싱 대상으로 한다. |W|의 크기는 접미어 시퀀스의 LCP 분석 결과에 근거하여 15정도를 사용할 수 있다. 이들 서브 시퀀스를 윈도우 서브 시퀀스라 부른다. DNA 시퀀스 S로부터 |S|개의 윈도우 서브 시퀀스를 추출하며, 시퀀스의 마지막 부분에 위치하는 길이가 |W|가 되지 못하는 윈도우 시퀀스에는 패딩 문자를 삽입하여 동일 길이로 한다. 이와 같은 동일 길이의 윈도우 서브 시퀀스를 이용한 인덱싱 기법은 인덱스의 크기를 감소시키고, 제 4장에서 보이는 단말 노드 검색을 간략화 시키는 효과를 가져 온다.

또한 보다 높은 인덱스 압축 효율을 구현하기 위하여 DNA 시퀀스가 소수의 문자만으로 구성된 시퀀스라는 점에 착안하여

시퀀스 내에 출현하는 각 문자를 8비트가 아닌 최소의 비트량으로 표현한다. 실제의 DNA 시퀀스에는 A, C, G, T 네 개의 문자 외에 출현 빈도가 높지 않은 몇 개의 와일드 카드 문자가 출현할 수 있다. 따라서 예를 들어 인덱싱 대상의 DNA 시퀀스에 7개 이하의 서로 다른 문자가 출현하는 경우, 각 문자에 3비트를 할당하여, 이를 구별 표현할 수 있다. 그림 3은 DNA 시퀀스에 출현하는 각 문자의 이진 표현 예이다. 여기에 사용된 '$'는 고정된 크기의 윈도우 서브 시퀀스를 생성하기 위하여 사용하는 패딩문자이다. 예를 들어, S='ACGACT'의 DNA 시퀀스에 대하여 각 시퀀스로부터 |W|=4의 윈도우 서브 시퀀스를 추출하여 그림 3에 따라 문자당 3비트를 사용하여 이를 이진 시퀀스로 변화하면 그림 4의 결과를 얻을 수 있다.

문 자	이진 표현
$	000
A	001
C	010
G	011
N	100
T	101
S	110
Y	111

그림 3. 각 출현 문자의 이진 표현 예

추출된 윈도우 서브 시퀀스	이진 변환된 윈도우 서브 시퀀스
ACGA	001010011001
CGAC	010011001010
GACT	011001010101
ACT$	001010101000
CT$$	010101000000
T$$$	101000000000

그림 4. DNA 시퀀스로부터 추출된 윈도우 서브 시퀀스의 이진 표현 예

3.3 인덱스 구축

인덱스는 이진 트라이 인덱스, 페이지 테이블, 단말 테이블의 3가지로 구성된다. 이진 트라이 인덱스는 윈도우 서브 시퀀스에 대한 기본 인덱스이며, 페이지 테이블은 디스크에 페이지 단위로 저장된 이진 트라이 인덱스의 페이지 단위 노드 간 연결 정보이고, 단말 테이블은 윈도우 서브 시퀀스의 실제 DNA 시퀀스내의 오프셋 정보이다. 인덱스 생성 알고리즘을 Algorithm 1에 보인다.

Algorithm 1: Index construction

```
Input : DNA sequence S, WindowSize |W|
Output : Binary Trie I, PageTable P, LeafTable L
Phase 1: Construct input sequences
  1. WS := extract_WindowSubsequences(S, |W|);
  2. sort(WS);
Phase 2: Build the index.
  1. MaxNode = computeMaxNode(pageSize);
  2. memPage = constructEmptyPage();
  3. for(i = 0; i < |WS|; i++) {
  4.     insertSequence(WSi, memPage);
  5.     if(isOverflow(memPage, MaxNode)) {
  6.         memToDisk(memPage, I, P, L);
  7.         reConstructPage(memPage);
  8.     }
  9. }
```

Algorithm 1은 DNA 시퀀스 S와 윈도우 사이즈 |W|를 입력으로 하여 BinaryTrie I, PageTable P, LeafTable L로 이루어진 이진 트라이 인덱스를 생성하는 방법이다. Algorithm 1의 동작을 간단히 설명하면 인덱스 생성을 위한 입력 시퀀스 생성 단계와 윈도우 서브 시퀀스를 순차적으로 인덱스에 삽입하는 두 개의 단계로 되어 있다. 첫 번째 단계에서는 DNA 시퀀스 S로부터 일정 길이 |W|의 모든 윈도우 서브 시퀀스를 추출하여 이 들을 오름차순으로 정렬한 윈도우 서브 시퀀스 리스트 WS를 생성한다(Line 1-2). 다음 단계에서는 1단계에서 만들어진 WS의 윈도우 서브 시퀀스를 이진 표현으로 변환하여 이진 트라이 구조에 삽입한다. 이진 데이터 표현의 이진 트라이는 처음에는 주기억 장치 memPage 영역에 생성되어 그 크기가 페이지 크기를 초과하면 디스크에 쓰이게 된다. 그 과정은 다음과 같다. 우선 페이지 크기 pageSize에 따라 compute MaxNode() 함수를 이용하여 한 페이지에 저장 가능한 최대 노드 레벨 수 혹은 최대 노드 수를 계산한 후, 주기억 장치 상에 페이지 영역 memPage를 생성한다(Line 1-2). 다음 insertSequence()함수는 이진 데이터 표현의 윈도우 서브 시퀀스 WSi를 memPage에 삽입하는 과정을 나타낸다(Line 4). 이때 새로이 삽입된 윈도우 서브 시퀀스 WSi에 의하여 memPage의 최대 노드 수 MaxNode를 초과하여 페이지 오버플로우가 발생하면, memToDisk()함수를 이용하여 주기억 장치내의 페이지 영역을 디스크에 기록하고, 해당 페이지 영역 memPage를 재구성한다(Line 5-9). 단, 디스크에 페이지 영역을 기록할 때, 해당 페이지에 대한 페이지 정보 및 단말 노드 정보를 페이지 테이블 P와 단말 테이블 L에 함께 기록한다.

3.2절에 사용된 예제를 사용하여 인덱스 생성 과정을 설명하면 다음과 같다. 다음의 그림 5는 그림 4의 윈도우 서브 시퀀스를 정렬하여, 처음으로 윈도우 서브 시퀀스 'ACGA(001 010 011 001)'를 삽입한 예를 보인다. 왼쪽 그림은 트라이를 노드 구조로 표현한 예이고, 오른쪽 그림은 실제의 이진 데이터 저장 방식을 나타낸다. 다음의 그림 6은 두 번째 서브 시퀀스 'ACT$(001 010 101 000)'를 추가로 삽입한 결과를 보인다. 이와 같이 트라이 구조에 새로운 임의의 길이의 이진 시퀀스를 삽입하면 새로이 삽입되는 시퀀스의 각 비트 정보는 트라이 노드 구조의 기존 경로를 따라 가거나, 새로운 에지를 생성하여 새로운 노드를 형성한다. 이와 같이 형성된 노드 구조는 2비트 노드 정보로 변환되어 이진 데이터로 저장된다. 그림 7은 그림 4의 예에서 보인 윈도우 서브 시퀀스를 모두 삽입한 결과의 트라이 구조를 나타내며, 그림 8은 그림 7의 이진 트라이 구조를 이진 데이터 형태로 저장한 예를 나타낸다.

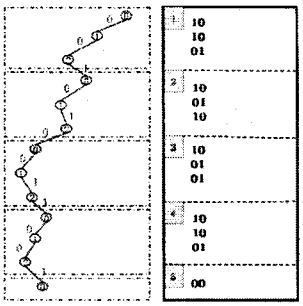

그림 5. 이진 트라이에 S의 첫 번째 서브 시퀀스 'ACGA' 삽입

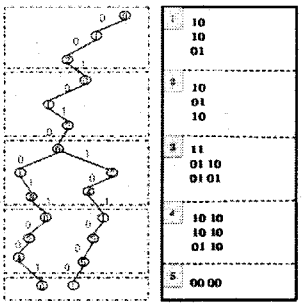

그림 6. 이진 트라이에 S의 두 번째 서브 시퀀스 'ACT$'삽입

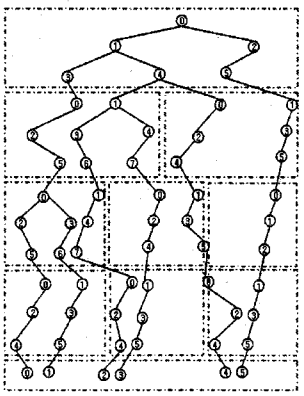

그림 7. 이진 트라이의 구성 예

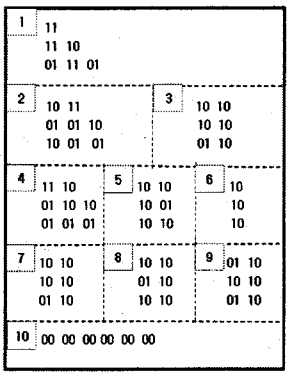

그림 8. 이진 트라이의 내부표현 예

#Page	Top	Bottom	Node	Addr
1	0	0	6	132
2	0	0	8	54
3	2	3	6	159
4	0	0	8	24
5	2	3	6	108
6	4	5	6	180
7	0	0	6	0
8	2	2	6	84
9	4	4	6	201
10	0	0	6	240

그림 9. 페이지 테이블

Offset
0
3
1
4
2
5

그림 10. 단말 테이블

그림 5, 6, 7, 8의 예에서 보이는 점선 영역은 디스크 상의 페이지 분할 영역을 나타낸다. 또한 그림 5, 6, 7에서 각 노드에 할당된 노드 번호는 디스크 페이지에 저장되는 노드의 저장 순서를 나타낸다. 예를 들어 페이지 크기를 16비트로 가정하여, 한 페이지에 저장할 수 있는 최대 노드 레벨 수를 3으로, 최대 노드 수를 8로 가정한 경우, 그림에서 보인 바와 같은 디스크 분할 결과를 얻게 된다. 그림 5의 예에서 각 2비트의 노드 정보는 페이지의 최대 노드 레벨 수에 의하여 최대 3개씩 각 페이지에 저장됨을 볼 수 있다. 이 후 각 페이지에 노드 정보가 추가되어 최대 노드 수를 초과하게 되면 각 페이지는 분리 저장되어 그림 7과 같은 페이지 분할 결과를 얻게 된다. 또한 여기에서 주의할 점은 트라이 구조를 페이지에 저장하는 경우, 한 페이지에 저장된 임의의 노드의 자식 노드가 다른 페이지에 나뉘어 저장되지 않도록 보장한다. 즉, 트라이를 구성하는 에지는 각 페이지의 수평 경계를 통과하여 생성될 수 있으나, 페이지의 수직 경계를 통과해서는 생성되지 못한다. 이 가정은 질의 처리 과정의 효율을 높이기 위하여 필요하다.

이와 같이 생성되는 트라이 인덱스는 디스크 페이지에 분할되어 저장되므로, 이 인덱스를 이용하여 임의의 경로를 검색하기 위해서는 노드와 노드 사이의 페이지 연결 상태를 나타내는 정보가 필요하다. 즉, 임의의 페이지에 저장되어 있는 노드로부터 직접 연결된 다음 노드가 어느 페이지에 저장되어 있는지를 알기 위해서는, 노드와 노드 사이의 페이지 연결 상태를 나타내는 정보가 부가적으로 필요하다. 이와 같은 페이지 연결 정보는 각 페이지에 유입된 에지의 수, 각 페이지로부터 나간 에지의 수 등으로부터 계산될 수 있다. 예를 들어 그림 8의 인덱스 구조의 페이지 교체를 위해서는 그림 9와 같은 페이지 정보가 필요하다. 여기에서 #page는 페이지의 번호를 나타내고, Top는 현재 페이지 레벨의 이전 페이지까지 유입된 에지의 총 수를 나타낸다. 또한, Bottom은 현재 페이지 레벨의 이전 페이지들로부터 나간 에지의 총 수를 나타내고, Node은 각 페이지에 기입된 노드 수를 나타내며, Addr은 각 페이지의 실제 기입 주소를 나타낸다. 또한 트라이 인덱스의 모든 단말 정보는 단말 테이블에 별도로 저장된다. 단말 테이블은 정렬되어 삽입된 각 윈도우 서브 시퀀스의 시퀀스 내 오프셋 정보를 나타내며, 그림 10에 그 예를 보인다.

4. 질의처리 방안

본 장에서는 제 3장에서 제안한 인덱싱 기법을 이용한 유사 검색 기법에 대해 논의한다. 유사 검색을 위하여 가장 기본적으로 사용되는 방식은 다이나믹 프로그래밍(DP) 기법이다. 비교 대상의 두 시퀀스 S와 Q에 대하여 DP 기법은 |Q|+1 개의 행과 |S|+1의 열을 갖는 2차원의 DP 테이블을 생성하여 두 시퀀스 사이의 최적의 거리(optimal distance)를 얻는다. 이 때 유사도를 계산하기 위하여 응용에 적합한 유사도 함수가 정의되어야 하며, 일반적으로 에디트 거리(edit distance) 등이 유사도 함수로 사용된다[5,9].

접미어 트리에서는 경로 운행에 의하여 모든 서브 시퀀스를 발견할 수 있다. 접미어 트리 기반의 유사 검색 알고리즘에서는 구조상 접미어 트리를 루트 노드로부터 깊이 우선 방식으로 운행하며 질의 시퀀스와 루트로부터 시작된 각 경로 상의 접미어 시퀀스에 대하여 DP 테이블을 생성함으로써 모든 유사 서브 시퀀스를 검색한다[6,9,10,19].

제안된 접미어 트라이 인덱스는 2비트의 노드 정보만을 저장할 뿐 노드와 노드를 연결하는 포인터 정보, 노드 레벨 정보, 경로 상의 해당 서브 시퀀스 정보를 별도로 저장하지 않는다. 따라서 인덱스 검색을 위해서는 해당 페이지의 노드 정보를 순차적으로 읽어 이 들의 정보를 얻어야 한다. 이 때 깊이 우선 방식을 사용하면 동일 페이지 내의 노드가 반복적으로 액세스 되며, 또한 디스크 상의 동일 페이지가 반복적으로 액세스 되는 단점이 있다.

이와 같은 특성을 고려하여 본 연구에서는 너비 우선 방식으로 이진 트라이를 운행하는 유사 검색 방식을 제안한다. Algorithm 2에 너비 우선 방식에 의하여 이진 트라이 인덱스 I를 탐색하여 질의 Q와 에디트 거리 T 이하의 유사 허용치를 만족하는 유사 서브 시퀀스들을 검색하는 알고리즘 Search-Trie를 보인다.

Search-Trie의 동작 과정을 간단히 설명하면 다음과 같다. 각 페이지와 페이지에 저장된 노드를 순차적으로 탐색하기 위하여 다음 2가지의 큐(queue) 구조를 이용한다. Qpagenumber는 탐색하게 될 페이지 정보를 담고 있으며, Qnode는 각 페이지에서 탐색될 노드 정보를 담고 있다. 우선, 트라이 인덱스의 루트 페이지의 번호를 Qpagenumber에, 루트 페이지에 대한 Qnode에 루트 노드를 각각 큐에 푸시한다(Line 1-2).

Algorithm 2: Search-Trie

```
Input : Binary Trie I, Query sequence Q, Toletance T,
        PageTable P, LeafTable L
Output : set of answers
1. push(Qpagenumber, Root_pageNumber);
2. push(Qnode[Root_pageNumber], RootNode);
3.
4. while (notEmpty(Qpagenumber)) do {
5.   pageNumber = pop(Qpagenumber);
6.   while (notEmpty(Qnode[pageNumber])) do {
7.     current_Node = pop(Qnode[pageNumber]);
8.     for each child node CNi of the current_Node do {
9.       moreVisit = TRUE,
10.      AppedBitString(CNi_Path, current_Node, CNi);
11.      if BitCount(CNi_Path) mod 3 == 0 {
12.        DPT_CNi = AddColumn(current_DPT, CNi_Path),
13.        Let dist be the last row value of the new
                                        added column;
14.        if dist <= T then FindAnswer(CNi, L),
             moreVisit = FALSE;
15.        else moreVisit = FurtherVisit(DPT_CNi);
16.      }
17.      if moreVisit {
18.        if terminal_Node(CNi) then
             FindCandidateAnswer(CNi, L);
           else {
19.          newPageNumber = getPageNumber(CNi, P);
20.          if newPageNumber != pageNumber
21.               and isEmpty(Qnode[newPageNumber]) then
22.            push(Qpagenumber, newPageNumber);
23.          push(Qnode[newPageNumber], CNi);
24.      }
25.    }
26.  }
27. }
28. }
```

전체 알고리즘은 탐색 대상의 페이지와 탐색 대상의 노드들에 대한 이중의 while 문으로 이루어져 있다. 외부 while 문(Line 4-28) 내의 Line 5는 탐색해야할 페이지 번호를 Qpagenumber로부터 팝업하는 과정을 나타낸다. 내부 while 문(Line 6-27) 내의 Line 6-7에서는 팝업된 페이지 번호의 Qnode로부터 탐색해야 할 노드 번호를 다시 팝업하여 현재 탐색 노드 current_Node로 한다. 다음, 현재 탐색 노드인 current_Node의 모든 차일드 노드들에 대하여 검색을 수행한다(Line 9-20). Line 9에 사용된 moreVisit는 이 후의 노드 검색을 계속 수행할지의 여부를 결정하는 변수로서 우선 그 값을 TRUE로 설정한다. Line 10의 AppendBitString()은 차일드 노드 CN_i의 방문에 의한 경로 정보를 추가하여 CN_i_Path를 생성하는 과정을 나타낸다. 이렇게 만들어진 경로 정보 CN_i_Path의 길이가 3의 배수라면(Line 11), 하나의 새로운 문자가 첨가된 것을 의미하므로 새로운 문자의 경로 정보를 이용하여 DP 테이블에 새로운 열을 추가하게 된다. Line 12의 함수 AddColumn(current_DPT, CN_i_Path)이 이 일을 수행하며, current_Node까지 생성된 DP 테이블 current_DPT를 사용하여 새로운 열이 추가된 DPT_CN_i를 생성하는 과정을 나타낸다. 여기에서 DPT_CN_i 테이블의 새로운 열의 마지막 요소 값을 dist로 한다(Line 13). 만약 dist의 값이 주어진 유사 허용치 T 보다 작거나 같으면 유사 서브 시퀀스가 검색되었으므로, 함수 FindAnswer()를 호출하여, 해당 노드 CN_i의 서브 트라이에 존재하는 모든 단말 노드로부터 시퀀스 정보(오프셋)를 얻게 된다. 그 후 그 경로에 대한 노드 탐색을 중지하기 위하여 moreVisit을 FALSE로 설정한다. 만약 dist의 값이 T 보다 큰 경우에는 함수 FurtherVisit()에 의하여 이 경로에 대한 노드 탐색을 계속할 것인지, 중지할 것인지의 여부를 결정한다. 이 때 moreVisit의 값이 TRUE이고 CN_i가 현재 탐색 중인 노드의 단말 노드인 경우에는 함수 FindCandidateAnswer()를 호출하여 해당 노드 CN_i를 후보 결과로 등록하고, 후처리 과정에 의하여 결과로서의 진위 여부

를 판별한다. 이 과정은 트라이 인덱스가 일정 길이의 윈도우 서브 시퀀스를 인덱싱 대상으로 하고 있으므로, 질의 길이가 윈도우 서브 시퀀스의 길이보다 큰 경우 발생할 수 있다. 그 외의 경우에는 차일드 노드를 Qnode에 푸시하여 그 경로에 대한 노드 탐색을 지속하도록 설정한다. Line 19-22의 과정은 차일드 노드가 현재 탐색 중인 페이지와 다른 페이지에 존재하는 경우를 위한 처리로서 새로운 페이지 번호를 Qpagenumber에 푸시하고, 새로운 페이지의 Qnode에 차일드 노드를 푸시한다.

Search-Trie 알고리즘에서는 질의 처리 과정 중, 이진 트라이의 중간 노드 CN_i에서 유사 서브 시퀀스가 검색되면, 해당 노드 CN_i의 서브 트라이에 존재하는 모든 단말 정보를 가져오는 과정이 필요하다. FindAnswer() 함수가 단말 정보를 단말 테이블 L을 이용하여 검색하는 역할을 담당한다. 이 함수는 해당 노드 CN_i의 이후 최좌측과 최우측 노드만을 방문하여 CN_i의 서브 트라이에 존재하는 모든 단말 정보를 가져온다. 즉, 이진 트라이에서 해당 노드 CN_i의 이후의 최좌측 노드만을 방문하여 얻어진 단말 노드를 LN_1이라 하고, 최우측 노드들만을 방문하여 얻어진 단말 노드를 LN_2라고 한다면 해당 노드 CN_i의 서브 트라이에 존재하는 모든 단말 정보는 LN_1과 LN_2사이에 존재하는 모든 단말 노드를 검색함으로써 얻을 수 있다. 이는 제안하는 기법에서는 윈도우 서브 시퀀스를 오름차순으로 정렬하여 이진 트라이내에 삽입하므로, 그림 9와 같이 구성된 단말 테이블 L에는 윈도우 서브 시퀀스가 삽입된 순서대로 시퀀스의 정보(오프셋)가 저장되어 있기 때문에 가능하다.

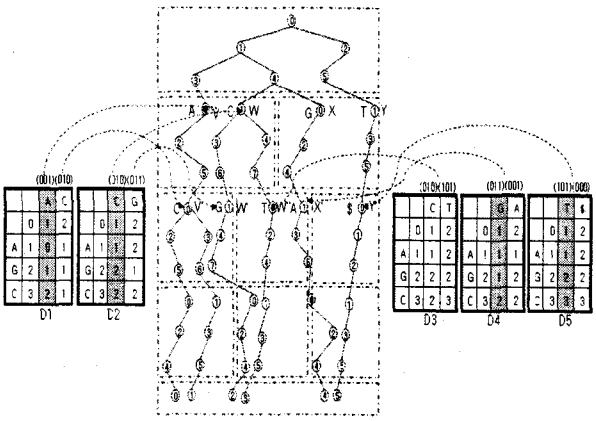

그림 10. 너비 우선 방식의 이진 트라이 운행 예

3장의 예제를 통해 트라이 인덱스를 이용하여 질의 시퀀스 'AGC'와 에디트 거리 T가 1이하인 유사 서브 시퀀스를 검색하는 과정을 그림 10에 보인다. 트라이 인덱스에서는 3비트가 하나의 문자를 나타내므로 트라이 인덱스의 경로를 따라 3개의 노드를 연속 방문 한 후, 기존의 DP 테이블에 새로운 열이 첨가되게 된다. 우선 너비 우선 방식에 의하여 루트 노드로부터의 순차적으로 v, w, x, y의 노드를 방문하여 'A(001)', 'C(010)', 'G(011)', 'T(101)의 문자가 각각 4개의 DP 테이블에 첨가되어 D1, D2, D4, D5가 얻어지게 된다. 이 때 생성된 DP 테이블의 열 값을 그림 10의 각 DP 테이블에 음영으로 처리하여 표시하였다. 다음 v', w', w'', x', y'의 노드를 방문하여 각 경로에서 얻어진 서브 시퀀스 'C(010)', 'G(011)', 'T(101)', 'A(001)', '$(101000)'에 의하여 5개의 DP 테이블 D1, D2, D3, D4, D5의 두 번째 열 값이 생성되게 된다. 여기에서 D3 테이블의 'C'열은 앞에서 작성된 D2의 'C'열과 공유되어 재사용된 것이다. DP 테이블에 새로운 열이 첨가될 때마다 새로이 첨가된 열의 마지막 요소 값이 주어진 유사 허용치보다 작거나 같은지의 조건이 만족되는지를 평가한다. 그림 10의 D1에서 현재 쌓은 'C'열의 마지막 요소 값이 1이므로 에디트 거리가 1인 서브 시퀀스가 검색되었음을 알 수 있다. 따라서 v' 노드 이하의 모든 단말 노드가 유사 검색 결과로 주어지게 된다. 또한 D3, D5에서는 현재 쌓은 열의 모든 요소 값이 1 보다 크므로 이 후의 경로에는 에디트 거리가 1 이하인 서브 시퀀스가 존재하지 않음을 알 수 있으며, 따라서 이 이상의 경로 검색을 중지한다. 이와 같은 과정에 의하여 이 후 트라이 상의 대상 경로를 너비 우선 방식으로 모두 순차 검색하게 된다. 따라서 본 방식에 의하여 트라이를 너비 우선 방식으로 운행하면, 각 디스크 페이지에 존재하는 노드 정보를 순차적으로 처리하게 되어 각 디스크 페이지는 최대 한번만 액세스 되는 효율적인 검색이 가능함을 알 수 있다.

5. 성능 평가

본 장에서는 실험에 의한 성능 분석을 통하여 제안하는 기법의 우수성을 규명한다. 본 실험에서는 GenBank[23]로부터 다운로드 받은 EST (Expressed Sequence Tags) 데이터를 이용한다. 실험에서 사용된 이 들 DNA 시퀀스에는 A, C, G, T의 네 종류의 문자와 몇 개의 와일드카드 문자가 출현한다. 여기에 '$'문자를 포함하여 8가지의 문자를 처리 대상으로 한다. 실험을 위한 플랫폼으로는 Redhat Linux FedoraCore 5 (Kernel Version 2.6.15)를 운영체제로 사용하고, 1GB의 주기억 장치, 40GB 디스크를 갖는 Athlon XP 1700+ (1.47GHz)의 PC를 사용한다.

제안된 기법의 우수성을 평가하기 위하여 인덱스의 크기 및 유사 서브 시퀀스 검색을 위한 질의 처리 시간을 기존에 방식과 실험을 통하여 비교 분석한다. Trie는 본 논문에서 제안한 트라이 인덱스 검색 방식을 사용한다. 성능 비교를 위한 기존 인덱싱 기법으로는 접미어 트리 Suffix를 사용한다. Suffix는 Top-Down Disk-based 접미어 트리 알고리즘을 사용한 접미어 트리를 사용한다(http://www.eecs.umich.edu/tdd). 실험 데이터로는 20.9M와 62.9M의 EST 데이터의 시퀀스 일부를 사용하였다. 질의 시퀀스는 실험 데이터로 사용되는 시퀀스 셋으로부터 임의의 방식으로 추출하였다. 추출된 질의 시퀀스를 사용하여 반복 실험 결과를 평균하였다.

실험 1에서는 본 논문에서 제안한 Trie의 인덱스 크기와 Suffix와 비교, 평가한다. 표 1에 데이터 크기의 변화에 따른 인덱스 크기 변화를 비교한 결과를 보인다. 본 실험에서는 Trie 인덱스의 페이지 크기를 4K로 설정하였다. Suffix 인덱스는 1차원 배열 형태로 저장되는 Tree 인덱스, LeafTable, RmostTable로 구성된다. Tree 인덱스는 내부 노드와 단말 노드 정보를 저장하고 있다. Tree 인덱스에 저장되지 않는 각 노드의 단말 여부와 Level 값을 계산하기 위한 정보를 LeafTable과 RmostTable에 각각 저장한다. Trie 인덱스는 이진 트라이 인덱스 BinaryTrie와 페이지 테이블 PageTable, 단말 테이블 LeafTable로 구성된다. Trie 인덱스 구성을 위하여 윈도우 크기를 15로 설정하였다. 표 1의 비교 결과로부터 Suffix, Trie의 두 방식 모두 데이터의 크기 변화에 따라 거의 선형적으로 인덱스 크기가 비례하여 증가함을 알 수 있다. 그러나 절대 크기를 비교 할 경우 Trie는 Suffix에 비하여 약 30%의 저장 공간만을 필요로 한다.

표 1. 인덱스 크기 비교 (단위 : Mbyte)

Data Size	Suffix				Trie			
	Tree	Leaf Table	RMost Table	Total	Binary Trie	Leaf Table	Page Table	Total
EST 20.9M	220.4	34	34	288.4	24	83.8	0.2	108
EST 62.9M	665.8	102.2	102.2	870.2	46.5	251.6	0.4	298.5

실험 2에서는 제안한 유사 검색 알고리즘 Search-Trie의 질의 처리 시간을 Suffix와 비교한다. 두 알고리즘 모두 실험 1에서 만든 인덱스를 사용한다. 질의 처리 시간은 DNA 시퀀스 S에서 질의 시퀀스 Q와 에디트 거리 T 이하의 유사 허용치를 만족하는 모든 유사 서브 시퀀스 S'를 검색하여 그 오프셋 값을 반환하는 총 시간을 의미한다.

그림 11에 20.9M와 62.9M의 DNA 시퀀스를 위한 질의 처리 시간을 각각 보인다. 여기에서 유사 허용치 T의 값으로 각 질의 길이의 10%에 해당하는 값을 사용하였다. 결과를 보면 Search-Trie의 질의 처리 시간이 Suffix에 비하여 약 1.7배에서 15배의 검색 속도가 빠름을 알 수 있다. 그러나 Search-Trie의 경우 질의 길이 40부터의 처리 시간이 크게 증가함을 보이고 있다. 이는 Trie 인덱스가 윈도우 사이즈가 15인 윈도우 서브 시퀀스를 사용하였기 때문으로 윈도우 사이즈보다 길이가 긴 질의 시퀀스의 검색 결과 중에서 후보 결과가 크게 증가하여 후처리 시간이 증가한 것이다.

다음 그림 12는 Search-Trie와 Suffx의 유사 허용치 변화에 따른 질의 처리 시간을 비교한 것이다. 여기에서 질의 길이 30의 질의를 사용하였으며, 유사 허용치는 1에서 4의 값을 사용하였다. 결과로부터 유사허용치가 증가하여도 Search-Trie가 Suffix에 비하여 약 1.8배에서 28배까지 속도가 빨라지고 있음을 알 수 있다. 그러나 유사 허용치가 4가 되면서 처리 시간이 크게 증가한다. 이는 그림 11의 결과와 같은 이유로 후보 결과의 증가에 따른 후처리 시간의 증가를 그 원인으로 들 수 있다.

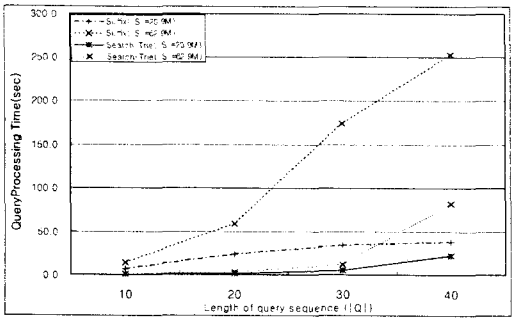

그림 11. 질의 길이 변화에 따른 질의 처리 시간 비교

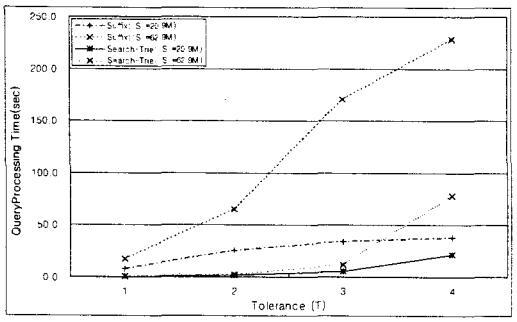

그림 13. 유사허용치 변화에 따른 질의 처리 시간 비교

6. 결론

본 논문에서는 효율적인 DNA 시퀀스 검색을 위한 인덱스 구조와 유사 검색 알고리즘을 제안하였다. 제안된 인덱스는 포인터 없이 트라이를 비트 스트링으로 표현하는 구조를 기본 구조로서 갖는다. 또한 DNA 시퀀스로부터 추출된 접미어의 LCP 길이가 비교적 짧다는 특성으로부터 DNA 시퀀스로부터 일정 길이의 윈도우 서브 시퀀스를 추출하여 이를 인덱스 대상으로 하였다. 이러한 구조의 장점은 저장 공간의 오버헤드를 크게 줄일 수 있다는 점과 전체 구조를 페이지 기반으로 형성시킬 수 있다는 점이다. 따라서 제안된 인덱스 구조를 사용함으로써 기존의 접미어 트리가 가지던 저장 공간, 검색 성능, DBMS와의 통합 등의 문제점들을 모두 해결할 수 있다.

또한, 제안된 인덱스를 이용하여 유사 검색을 효과적으로 처리하는 질의 처리 알고리즘을 제시하였다. 유사 검색 알고리즘은 다이나믹 프로그래밍 기법에 근거하여 이진 트라이를 루트로부터 너비 우선(breadth-first) 방식으로 운행하며 경로 상에 존재하는 유사 서브 시퀀스를 효율적으로 검색해낸다.

제안된 기법의 우수성을 검증하기 위하여, 실험을 통한 성능 평가를 수행하였다. 실험 결과에 의하면, 제안된 인덱스는 기존의 접미어 트리와 비교하여 약 30%의 저장 공간을 가지고도 기존의 접미어 트리에 비하여 약 1.8배에서 28배까지의 검색 성능의 개선 효과를 나타내는 것으로 나타났다.

참고문헌

[1] C. Gibas and P. Jambeck, Developing Bioinformatics Computer Skills, O'Reilly and Associates Inc., 2001.

[2] T. Smith and M. Waterman, "Identification of Common Molecular Subsequences," Journal of Molecular Biology, 147, pp. 195-197, 1981.

[3] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic local alignment search tool," Journal of Molecular Biology, Vol. 215, No. 3, pp. 403-410, 1990.

[4] S. Altschul, T. Madden, A. Schaffer, J. Zhang, W. Miller, and D. Lipman, "Gapped BLAST and PSI-BLAST: A New Generation of Protein Database Search Programs," Nucleic Acids Research, Vol 25, No. 17, pp. 3389-3402, 1997.

[5] G. A. Stephen, String Searching Algorithms, World Scientific Publishing, 1994.

[6] E. Ukkonen, "Approximate string matching over suffix trees," In Proceedings of Combinatorial Pattern Matching (CPM93), pp. 228-242, 1993.

[7] A. L. Delcher, S. Kasif, R. D. Fleischmann, and J. Peterson, O. White, and S. L. Salzberg, "Alignment of whole genomes," Nucleic Acids Research, 27, pp. 2369-2376, 1999.

[8] S. Kurtz, J. Choudhuri, E. Ohlebusch, C. Schleiermacher, J. Stoye, and R. Giegerich, "REPuter: the manifold applications of repeat analysis on a genome scale," Nucleic Acids Research, Vol. 29, No. 22, pp. 4633-4642, 2001.

[9] E. Hunt, M. P. Atkinson and R. W. Irving, "Database indexing for large DNA and protein sequence collections," The VLDB Journal, Vol. 11, No. 3, pp. 256-271, 2002.

[10] G. Navarro and R. Baeza-Yates, "A Hybrid Indexing Method for Approximate String Matching," J. of Discrete ALgorithms, Vol. 1, No. 1, pp.205-239, 2000.

[11] S. Kurtz, "Reducing the Space Requirement of Suffix Trees," Softw. Pract. Exp., Vol 29, pp. 1149-1171, 1999.

[12] R. Giegerich, S. Kurtz, and J. Stoye, "Efficient Implementation of Lazy Suffix Trees," Softw. Pract. Exp., Vol 33, pp. 1035-1049, 2003.

[13] K. Sadakane and T. Shibuya, "Indexing huge genome sequences for solving various problems," In Proceedings of the 12th Genome Informatics, pp. 175-183, 2001.

[14] H. Wang et al., "BLAST++: A Tool for BLASTing Queries in Batches," In Proceedings First Asia-Pacific Bioinformatics Conference, pp. 71-79, 2003.

[15] H. E. Williams and J. Zobel, "Indexing and Retrieval for Genomic Databases," IEEE TKDE Vol. 14, No. 1. pp. 63-78, 2002.

[16] A. Califano and I. Rigoutso, "FLASH: A Fast Look-up Algorithm for String Homology," In Proceedings of Intelligent System Conference for Morecular Biology, pp. 56-64, 1993.

[17] C. Fondrat and P. Dessen, "A Rapid Access Motif database(RAMdb) with a search algorithm for the retrieval patterns in nucleic acids or proteun databanks," Computer Applications in the Biosciences. Vol. 11, No.3, pp. 273-279, 1995.

[18] T. Kahveci and A. K. Singh, "An Efficient Index Structure for String Databases," In Proceedings of the 27th VLDB Conference, pp. 351-360, 2001.

[19] C. Meek, J. M. Patel, and S. Kasetty, "OASIS: An Online and Accurate Technique for Local-Alignment Searches on Biological sequences," In Proceedings of the 29th VLDB Conference, pp. 920-921, 2003.

[20] S. Tata, R. Hankins, and J. Patel, "Practical Suffix Tree Construction," In Proceedings of the 30th VLDB Conference, pp. 36-47, 2004.

[21] E. Horowitz, S. Sahni, and S. Anderson-Freed, Fundamentals of Data Structures in C, Computer Science Press, 1993.

[22] H. Shang and T. H. Merrett, "Tries for approximate string matching," IEEE Trans. on Knowlege and Data Engineering, Vol. 8, No. 4, pp. 540-547, 1996.

[23] http://www.ncbi.nlm.nih.gov