DNA 시퀀스 데이타베이스를 위한 실용적인 유사 서브 시퀀스 검색 기법

(A Practical Approximate Sub-Sequence Search Method for DNA Sequence Databases)

원 정 임 † 홍 상 균 †† 윤 지 희 ††† 박 상 현 †††† 김 상 욱 †††††

(Jung-Im Won) (Sang-Kyoon Hong) (Jee-Hee Yoon) (Sang-Hyun Park) (Sang-Wook Kim)

요 약 유사 서브 시퀀스 검색은 분자 생물학 분야에서 사용되는 매우 중요한 연산이다. 본 논문에서는 대규모 DNA 시퀀스 데이타베이스를 처리 대상으로 하여 효율성과 정확도를 보장하는 실용적인 유사 서브 시퀀스 검색 기법을 제안한다. 제안된 기법은 이진 트라이를 인덱스 구조로 채택하여 DNA 시퀀스로부터 추출한 일정 길이의 윈도우 서브 시퀀스를 인덱싱 대상으로 한다. 유사 서브 시퀀스 검색 알고리즘은 기본적으로 다이나믹 프로그래밍 기법에 근거하여 이진 트라이를 루트로부터 너비 우선(breadth-first) 방식으로 운행하며, 경로 상에 존재하는 모든 유사 서브 시퀀스를 검색해 낸다. 그러나 질의 길이가 윈도우의 크기보다 큰 일반적인 경우에는 질의를 일정 길이의 서브 시퀀스로 분해하여 각 서브 시퀀스에 대하여 유사 서브 시퀀스 검색을 수행한 후, 후처리 과정에 의하여 정확도에 손상 없이 이 들 결과를 결합하는 분할 질의 처리 방식을 채택한다. 제안된 기법의 우수성을 검증하기 위하여, 실험을 통한 성능 평가를 수행한다. 실험 결과에 의하면, 제안된 인덱스 기법은 접미어 트리에 비하여 약 40%의 작은 저장 공간을 가지고도 약 4~17배의 검색 성능의 개선 효과를 나타낸다. 또한 분할 질의 처리 방식에 의한 유사 서브 시퀀스 검색 알고리즘은 질의 길이가 긴 경우에도 효율적으로 동작하여 Suffix와 Smith-Waterman 알고리즘에 비하여 각각 수배에서 수십배의 검색 성능의 개선 효과를 나타낸다.

키워드 : DNA 시퀀스 데이타베이스, 유사 서브 시퀀스 검색, 인덱싱, 트라이, 접미어 트리

Abstract In molecular biology, approximate subsequence search is one of the most important operations. In this paper, we propose an accurate and efficient method for approximate subsequence search in large DNA databases. The proposed method basically adopts a binary trie as its primary structure and stores all the window subsequences extracted from a DNA sequence. For approximate subsequence search, it traverses the binary trie in a breadth-first fashion and retrieves all the matched subsequences from the traversed path within the trie by a dynamic programming technique. However, the proposed method stores only window subsequences of the pre-determined length, and thus suffers from large post-processing time in case of long query sequences. To overcome this problem, we divide a query sequence into shorter pieces, perform searching for those subsequences, and then merge their results. To verify the superiority of the proposed method, we conducted performance evaluation via a series of experiments. The results reveal that the proposed method, which requires smaller storage space, achieves 4 to 17 times improvement in performance over the suffix tree based method. Even when the length of a query sequence is large, our method is more than an order of magnitude faster than the suffix tree based method and the Smith-Waterman algorithm.

Key words : DNA sequence database, approximate subsequence search, indexing, trie, suffix tree

· 본 논문은 2005년 정부재원(교육인적자원부 학술연구조성사업비)으로 한국학술진흥재단(KRF-2005-206-D00015)과 2006년도 정부(과학기술부)의 재원으로 한국과학재단(No. R01-2006-000-11106-0), 제주대학교를 통한 정보통신부 및 정보통신 진흥원의 대학 IT 연구센터 지원사업(IITA-2005-C1090-0502-0009)의 지원을 받았습니다.

† 정 회 원 : 한양대학교 정보통신학부 교수
jiwon@hanyang.ac.kr
†† 학생회원 : 연세대학교 컴퓨터과학과
kyoons@gmail.com
††† 종신회원 : 한림대학교 정보통식공학부 교수
jhyoon@hallym.ac.kr
†††† 종신회원 : 연세대학교 컴퓨터과학과 교수
sanghyun@cs.yonsei.ac.kr
††††† 종신회원 : 한양대학교 정보통신학부 교수
wook@hanyang.ac.kr

논문접수 : 2006년 10월 18일
심사완료 : 2007년 1월 24일

1. 서 론

DNA 시퀀스는 모든 생물의 생명 특성을 결정하는 코드로서 A, C, G, T의 네 가지 문자들로 구성된 스트링으로 표현된다. 분자 생물학에서 어떤 DNA 시퀀스를 해석하기 위한 가장 기본적인 방법은 그것과 염기 배열이 유사한 다른 DNA 시퀀스의 특성을 이용해 구조 및 기능 등을 유추하는 것이다[1].

DNA 시퀀스를 대상으로 하는 유사 서브 시퀀스 검색(approximate subsequence match) 문제는 다음과 같이 정의된다[2,3]. DNA 시퀀스 S와 질의 시퀀스 Q, 유사 허용치(tolerance) T가 주어지면, Q와 유사 허용치가 T 보다 작거나 같은 서브 시퀀스 S′을 S 내에서 검색해 낸다. 이때 Q와 S′ 사이의 유사도를 계산하기 위하여 응용 목적에 적합한 유사도 함수(similarity function)를 정의하여 사용하게 되며, 에디트 거리(edit distance) 등을 예로 들 수 있다.

DNA 시퀀스 검색을 위하여 일반적으로 사용되는 방식은 데이타베이스 전체를 검색 대상으로 하는 순차 검색 기법이다[4]. BLAST [5,6]는 DNA 시퀀스 검색 연산을 위하여 가장 널리 사용되는 표준 도구이다. BLAST는 휴리스틱스 기반 알고리즘에 의하여 고속의 시퀀스 검색 기능을 제공하지만, 검색 결과의 완전한 정확도를 보장하지는 못한다. 유사 서브 시퀀스 검색 연산의 정확성을 보장할 수 있는 대표적인 알고리즘으로 Smith-Waterman 알고리즘[4]를 들 수 있다. 그러나, 이 방식은 많은 계산량을 필요로 하여 속도가 느리므로 실제적인 사용에 있어 제한적이다.

최근의 DNA 시퀀스 데이타베이스 규모의 급격한 증가 추세를 고려할 때, DNA 시퀀스 검색 연산을 보다 효과적으로 지원할 수 있는 인덱싱 및 질의 처리 기술이 요구된다. 접미어 트리(suffix tree)[7]는 DNA 시퀀스 검색을 위한 좋은 인덱스 구조로 알려져 왔다. 접미어 트리는 주어진 시퀀스의 접미어에 해당되는 서브 시퀀스들을 트리 형태로 구성한 것으로서 질의와 정확히 일치하는 서브 시퀀스를 고속으로 검색한다. 최근 효율적인 접미어 트리 구성 방식 및 접미어 트리 기반의 유사 서브 시퀀스 검색 알고리즘들에 관한 연구가 활발히 이루어지고 있다[8-13]. 그러나 접미어 트리는 그 구조적 특성으로 인하여 다음과 같은 문제점을 갖는다.

① 접미어 트리는 일반적으로 매우 큰 저장 공간을 필요로 하여, 그 크기는 데이타베이스 크기의 수 십배에 이른다[14-16].

② 디스크 액세스 패턴이 지역적이지 못하여 검색 성능이 떨어진다. 접미어 트리의 크기가 클수록 이 트리를 순회하는 시간이 길어지며, 따라서 전체 검색 성능이 떨어진다[12].

③ 접미어 트리는 그 구조적 특성 상 페이지 단위로 구현하기가 어렵다. 따라서 모든 데이타를 페이지 단위로 구현하는 DBMS와 밀 결합(seamless integration)에 어려움이 있다[7,17].

본 논문에서는 이와 같은 접미어 트리의 문제점들을 해결하는 DNA 시퀀스 검색을 위한 새로운 인덱스 구조를 제안하고, 이를 기반으로 하는 효율적인 질의 처리 방식을 제안한다. 본 연구에서 제안하는 인덱스는 트라이(trie)[7,18]를 기본 구조로 하여 포인터 없이 트라이를 비트 스트링으로 표현하는 방식을 채택한다. 또한 DNA 시퀀스를 구성하는 모든 문자의 시작 위치로부터 일정 길이의 서브 시퀀스(윈도우 서브 시퀀스)를 추출하여 이를 인덱싱의 대상으로 한다. 이는 DNA 시퀀스의 접미어들의 최대 공통 접두어(LCP: Longest Common Prefix)의 크기가 평균적으로 매우 작다는 성질을 이용한 결과로서 접미어 대신에 비교적 작은 크기의 윈도우 서브 시퀀스를 인덱싱 대상으로 한다.

제안된 인덱스를 기반으로 하는 유사 서브 시퀀스 검색 알고리즘은 다이나믹 프로그래밍 기법을 사용하여 이진 트라이 경로 상에 존재하는 모든 유사 서브 시퀀스를 검색한다. 트라이 인덱스 탐색 방식으로서 너비 우선 방식을 채택하여 인덱스를 구성하는 각 페이지 데이타에 대하여 단 한 번의 디스크 액세스에 의한 단 한 번의 순차적 처리에 의하여 모든 경로 상의 유사 서브 시퀀스를 검색해낸다. 그러나 제안된 인덱스는 일정 길이의 윈도우 서브 시퀀스만을 인덱싱 대상으로 하고 있으므로 질의 길이가 커지는 경우 후처리 과정에 의한 질의 처리 시간이 증가하게 된다. 따라서 본 연구에서는 질의의 길이에 따라 질의를 적절히 분할하여 처리하는 분할 질의 처리 기법을 사용하여 질의 처리 성능을 최적화한다.

다양한 실험을 통하여 제안된 인덱스 및 유사 검색 기법의 성능을 기존의 방식과 비교하여 정량적으로 검증한다. 실험 결과에 의하면, 제안된 이진 트라이 인덱스 기법은 접미어 트리에 비하여 약 40%의 작은 저장 공간을 가지고도 수배에서 수십배의 검색 성능의 개선 효과를 나타내며, 특히 질의 길이가 긴 경우에도 효율적으로 동작함을 보인다.

본 연구의 주요 기여도는 다음과 같다. 1) DNA 시퀀스의 특성을 고려한 효율적인 인덱싱 기법을 제안하였다.: 이진 트라이 인덱스는 접미어 트리에 비하여 작은 저장 공간을 사용하며, 진정한 의미의 디스크 페이지 기반의 인덱스 구조를 가지고 있어 DNA 인덱싱 기법으로 가장 잘 알려진 접미어 트리를 대신하여 접미어 트리의 응용 분야에 활용이 가능하다. 2) 인덱스 기반의

효율적인 유사 서브 시퀀스 검색 기법을 제안하였다.: 일반적으로 질의 길이가 길거나 유사 허용치가 커지는 경우, 접미어 트리 등 인덱스 기반의 유사 서브 시퀀스 검색은 효율을 보장하기 어렵다는 생각이 받아들여지고 있다. 본 연구에서는 실용 규모의 DNA 데이타베이스에 대하여 질의 길이가 길고 유사 허용치가 큰 경우에도 인덱스 기반의 유사 서브 시퀀스 검색이 효율적으로 이루어짐을 보인다.

본 논문의 구성은 다음과 같다. 제 2장에서는 본 연구의 배경으로서 DNA 시퀀스 검색과 관련된 기존의 연구들을 간략히 소개한다. 제 3장에서는 제안하는 인덱싱 방법을 제시하고, 제 4장에서는 이를 이용한 유사 서브 시퀀스 검색 방법을 제안한다. 제 5장에서는 실험에 의한 성능 평가를 통하여 제안하는 기법의 우수성을 규명하고, 제 6장에서는 결론을 내린다.

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[4]를 들 수 있다. SW 알고리즘은 다이나믹 프로그래밍(dynamic programming)기법에 의하여 두 시퀀스 S와 Q 사이에 최적의 부분 정렬(optimal local alignment)을 찾는다. 그러나 이 방식은 두 시퀀스의 길이의 곱에 비례하는 계산량(O(|Q|×|S|))을 필요로 하여 속도가 느린 단점이 있다.

BLAST[5,6]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 데이타베이스가 주기억 장치에 적재되어야 하며, 검색 속도가 데이타베이스 사이즈에 비례하고, 워드의 고정 길이에 따라 검색 결과의 정확도가 영향을 받는 점 등이 문제점으로 지적된다.

DNA 시퀀스 데이타베이스에 대하여 사전에 인덱스를 구성, 활용함으로써 DNA 시퀀스 검색의 속도 향상을 기대할 수 있다. 인덱스 기반의 DNA 시퀀스 검색 기법은 역 인덱스(inverted index) 방식[3,19,20], 다차원 인덱스 방식[21], 영속 트리 방식[9,11,12] 등으로 분류할 수 있다.

정보 검색 분야에 활용되는 역 인덱스(inverted index)를 기반으로 하는 인덱싱 기법으로 참고문헌 [3,19,20]의 방식을 들 수 있다. 시퀀스 데이타베이스 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. 참고문헌 [3]에서는 특히 압축 인덱스 구조를 사용하여 인덱스 파일이 과다하게 커지는 단점을 보완하고 있으나, 검색 결과의 정확도가 떨어지는 점 등이 단점으로 지적되고 있다.

참고문헌 [21]에서는 웨이블렛(wavelet) 변환에 의하여 시퀀스 데이타베이스 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 데이타베이스의 검색 공간을 축소시킬 수 있는 효율적인 방식으로 간주될 수 있다.

영속 트리 기반의 대표적인 인덱싱 방식으로서 접미어 트리(suffix tree)[7]을 들 수 있다. 기존에는 디스크 상에 주기억장치 용량 이상인 대규모의 접미어 트리를 구성하는데 어려움이 있었으나, 최근 참고문헌 [9]는 디스크 분할 저장 방식(partitioned suffix tree)에 의한 접미어 트리 구성 방안을 제안하고 있으며, 참고문헌 [22]는 Top-Down Disk-based 방식의 효율적인 접미어 트리 구성 방안을 제안하고 있다. 이 들 접미어 트리를 기반으로 하는 유사 서브 시퀀스 검색 알고리즘들은 접미어 트리를 루트 노드로부터 깊이 우선 방식으로 운행하며 각 경로 상의 접미어 시퀀스에 대하여 DP 테이블을 생성함으로써 모든 유사 서브 시퀀스를 검색한다 [11,13,16,19]. 참고문헌 [11]에서는 다이나믹 프로그래밍 A*-탐색 기법[23]을 적용하여 유사도 우선 순으로 검색 결과를 반환하는 유사 검색 방식을 제안하고 있다. 그러나 이와 같은 접미어 트리 기반의 유사 검색 알고리즘은 대규모 인덱스 검색 공간의 문제로 인하여 실용적이지 못하다. 최근 이와 같은 인덱스 검색 공간의 문제를 해결하기 위한 방안의 하나로서 질의 분할 처리 기법이 제안되었다[16]. 참고문헌 [16]에서는 주어진 질의 시퀀스를 작은 길이의 서브 질의로 분해하여 각 서브 질의에 대하여 작은 T 값을 대상으로 유사 검색을 수행한 후, 그 결과를 결합하는 분할 처리 기법을 제안하고 그 타당성을 보이고 있다.

한편 접미어 트리의 공간 문제를 해결하기 위한 방안으로 suffix array[24], 혹은 compact suffix array[25], compressed suffix array[26]를 인덱스 구조로 사용할 수 있다. 이 들 인덱스 구조는 접미어 트리의 축약형으로 접미어 트리의 단말 노드 정보만을 정렬하여 압축 저장한 형태이므로, 접미어 트리에 비하여 작은 저장 공간으로 구현 가능하다. 그러나 이 들 인덱스 구조를 사용하는 경우, 접미어 트리에 비하여 공간 활용도가 높아지는 만큼 트리 탐색 시간이 길어지는 단점이 있다.

3. 인덱싱 방안

본 장에서는 대규모 DNA 시퀀스 데이타베이스를 대상으로 하여 효율적인 DNA 시퀀스 검색을 지원하기 위한 인덱싱 방안에 대하여 논의한다. 제 3.1절에서는 인덱스의 기본 구조로 사용하는 이진 트라이 구조를 설명하고, 제 3.2절에서는 인덱싱 대상이 되는 DNA 시퀀스 상의 윈도우 서브 시퀀스의 추출 방식에 대하여 설명한다. 제 3.3절에서는 이 들을 이용한 인덱스 구성 알고리즘을 보인다.

3.1 이진 트라이

트라이[7,18]는 노드가 정보를 가지지 않고 에지에 데이타가 저장되는 트리 구조로서, 각 에지는 하나의 문자를 가지며 루트로부터 단말까지의 경로는 인덱싱의 대상이 되는 하나의 키에 해당된다. 트라이를 구현하는 가장 직접적인 방법은 각 노드를 $|\Sigma|$개의 포인터를 저장하는 배열로 나타내는 것이다. 여기서, Σ는 응용이 대상으로 하는 문자 집합, 즉 알파벳을 나타낸다. 이 방법을 사용하면 검색 연산을 효율적으로 수행할 수 있지만 노드에 널(NULL) 포인터가 많아져서 기억 공간을 낭비하는 경향이 있다. 일반적으로 $|\Sigma|$이 커질수록 기억 공간의 낭비가 커지며, 루트 근처에 있는 노드보다는 단말 근처에 있는 노드가 기억 공간을 더 많이 낭비하게 된다.

트라이의 노드를 링크된 리스트로 구현할 수도 있다. 즉, 각 노드에 바로 오른쪽 형제 노드에 대한 포인터와 가장 왼쪽 자식 노드에 대한 포인터를 저장함으로써 조건을 만족시키는 자식 노드로 손쉽게 이동할 수 있다. 이 방법에서는 꼭 필요한 기억 공간만이 노드에 할당되기 때문에 기억 공간의 낭비가 최소화 되는 장점이 있지만, 조건을 만족시키는 자식 노드가 상수 시간(constant time)에 선택될 수 없기 때문에 검색 연산의 효율이 다소 저하되는 단점이 있다.

꼭 필요한 기억 공간만을 사용하면서도 검색 연산을 효율적으로 수행할 수 있는 방안으로 포인터 없는 이진 트라이(pointerless binary trie)[27]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.

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 이들 접미어들이 많은 공통 접두어를 가질 때 좋은 압축 효과를 갖는다. DNA 시퀀스는 A, C, G, T 네 개의 문자로 구성된 매우 긴 시퀀스로 볼 수 있다. 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 1에 DNA 시퀀스로부터 추출한 접미어

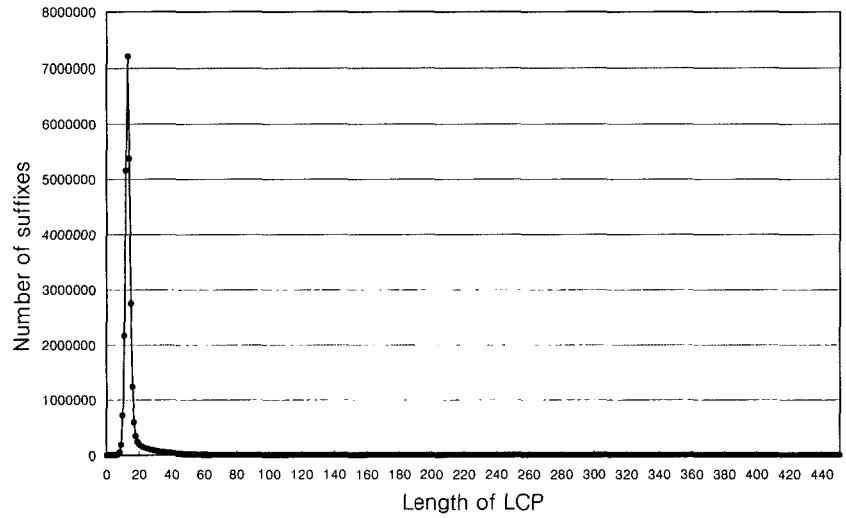

그림 1 DNA 시퀀스로부터 추출된 접미어 시퀀스들의 LCP 분포

들의 최대 공통 접두어 LCP의 크기 분포를 보인다. 분석 데이타로는 21번 human chromosome의 일부인 28.6Mbp의 DNA 시퀀스를 사용하였다. X축은 LCP 크기를 나타내며, Y축은 각 LCP 크기를 갖는 접미어의 총 개수를 나타낸다. 이 그림으로부터 접미어들은 평균 LCP 크기가 15이고, 최대 LCP 크기가 451이며, LCP의 크기 = 13에서 최대 출현 회수(약 720만개)를 가짐을 알 수 있다. 또한 LCP 크기의 분포가 11에서 15사이에 집중되어 있어 LCP의 크기 15까지의 총 접미어의 누적 비율은 전체의 82.6%까지 이르고 있음을 알 수 있다.

이와 같은 DNA 시퀀스의 특성을 고려하여 본 연구에서는 DNA 시퀀스의 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. 금후 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 윈도우 서브 시퀀스를 이용한 인덱싱 기법은 인덱스의 크기를 감소시키고, 제 4.1절에서 보이는 단말 노드 검색을 간략화 시키는 효과를 가져온다.

또한 본 연구에서는 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 11개의 와일드 카드 문자가 출현할 수 있다. 예를 들어 N은 A, C, G, T 중 임의의 하나의 문자를 나타내며, B는 A가 아닌 C, G, T 중 임의의 하나의 문자를 의미한다. 따라서 인덱싱 대상의 DNA 시퀀스에 7개 이하의 서로 다른 문자가 출현하는 경우 각 문자에 3비트를 할당하여, 이를 구별 표현할 수 있다. 이와 같은 방식에 의하여 DNA 시퀀스를 축약된 이진 시퀀스로 변환하여 트라이를 구성하면 공통 접두어가 생성될 확률이 더욱 높아지며, 따라서 생성된 트라이 인덱스 구조로부터 더욱 높은 압축율을 기대할 수 있다.

다음의 그림 2에 DNA 시퀀스에 출현하는 각 문자의 이진 표현 예를 보인다. 여기에서 사용된 '$' 문자는 고정 크기의 윈도우 서브 시퀀스를 생성하기 위하여 사용하는 패딩 문자를 나타낸다. 예를 들어, S='ACGACT'

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

그림 2 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 3 DNA 시퀀스로부터 추출된 윈도우 서브 시퀀스의 이진 표현 예

의 DNA 시퀀스에 대하여 각 시퀀스로부터 |W|=4의 윈도우 서브 시퀀스를 추출하여 그림 2의 방식에 의하여 각 문자당 3비트를 할당하여 이를 이진 시퀀스로 변환하면 그림 3과 같은 결과를 얻을 수 있다.

3.3 인덱스 구축

제안하는 인덱스는 이진 트라이 구조를 기본으로 사용하며, DNA 시퀀스로부터 추출된 모든 윈도우 서브 시퀀스를 인덱싱 대상으로 한다. 인덱스는 이진 트라이 인덱스, 페이지 테이블, 단말 테이블의 3가지 요소로 이루어진다. 이진 트라이는 윈도우 서브 시퀀스를 이진 트라이 구조로 구성한 기본 인덱스 구조이며, 페이지 테이블은 이진 트라이 인덱스 페이지의 노드 간 연결 정보를 나타내고, 단말 테이블은 윈도우 서브 시퀀스의 DNA 시퀀스 내의 오프셋 정보를 나타낸다.

인덱스 생성 알고리즘을 다음의 Algorithm 1에 보인다. Algorithm 1은 DNA 시퀀스 S와 윈도우 사이즈 |W|를 입력으로 받아, binary trie I, page table P, leaf table L로 이루어진 이진 트라이 인덱스를 생성한다. Algorithm 1의 동작 과정을 간단히 설명하면 다음과 같다. 1단계는 인덱스 생성을 위한 입력 시퀀스 생성 단계를 나타낸다. DNA 시퀀스 S로부터 일정 길이 |W|의 모든 윈도우 서브 시퀀스를 추출하여 이 들을 오름차순으로 정렬한 윈도우 서브 시퀀스 리스트 WS를 생성한다(Lines 1~2). 다음 2단계는 WS의 윈도우 서브 시퀀스를 순차적으로 인덱스에 삽입하는 과정을 나타내며, 윈도우 시퀀스는 이진 표현으로 변환되어 이진 트라이 구조에 삽입된다. 이 때 윈도우 서브 시퀀스는 정렬된

순서에 의하여 삽입되어지는 것을 가정한다. 이는 이진 트라이 인덱스 구조를 단일 방향으로 증가시키는 효과를 가지게 하여 디스크 상의 페이지 기반 인덱스 구성을 가능하게 한다. 이와 같이 생성되는 이진 데이타 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 computeMaxNode() 함수를 이용하여 한 페이지에 저장 가능한 최대 노드 레벨 수 혹은 최대 노드 수를 계산한 후, 주기억 장치 상에 페이지 영역 memPage를 생성한다(Lines 3~4). 다음 insertSequence()함수는 이진 데이타 표현의 윈도우 서브 시퀀스 WS_i를 memPage에 삽입하는 과정을 나타낸다(Line 6). 이때 새로이 삽입된 윈도우 서브 시퀀스 WS_i에 의하여 memPage의 최대 노드 수 maxNode를 초과하여 페이지 오버플로우가 발생하면, memToDisk()함수를 이용하여 주기억 장치내의 페이지 영역을 디스크에 기록하고, 해당 페이지 영역 memPage를 재구성한다(Lines 7~9). 단, 디스크에 페이지 영역을 기록할 때, 해당 페이지에 대한 페이지 정보 및 단말 노드 정보를 페이지 테이블 P와 단말 테이블 L에 함께 기록한다.

Algorithm 1: Index construction

```
Input : DNA sequence S, window size |W|
Output : binary trie I, page table P, leaf table L
Step 1: Construct input sequences
1.  WS := extractWindowSubsequences(S, |W|);
2.  sort(WS);

Step 2: Build the index
3.  maxNode = computeMaxNode(pageSize);
4.  memPage = constructEmptyPage();
5.  for(i = 0; i < |WS|; i++)
6.      insertSequence(WSi, memPage);
7.      if(isOverflow(memPage, maxNode))
8.          memToDisk(memPage, I, P, L);
9.          reconstructPage(memPage);
```

제 3.2절에 사용된 예제를 사용하여 인덱스 생성 과정을 간단히 설명하면 다음과 같다. 다음의 그림 4는 그림 3의 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 데이타 저장 방식을 나타낸다. 다음의 그림 5는 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 데이타로 저장된다. 그림 6은 그림 3의 예에서 보인 윈도우 서브 시퀀스를 모두 삽입한 결과의 트라이 구조를 나타내며, 그림 7은 그림 6의 이진 트라이 구조를 이진 데이타 형태로 저장한 예를 나타낸다.

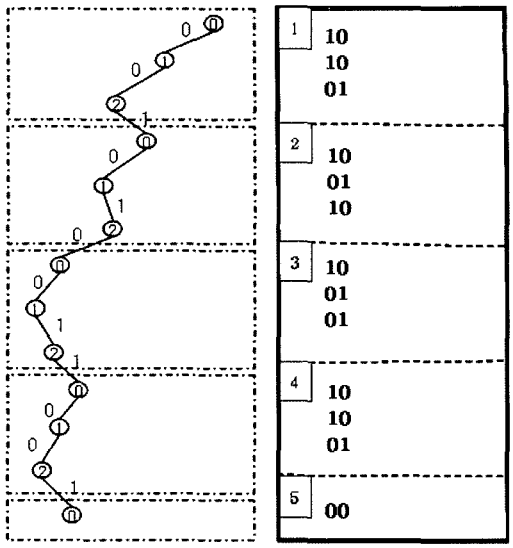

그림 4 이진 트라이에 S의 첫 번째 서브 시퀀스 'ACGA' 삽입

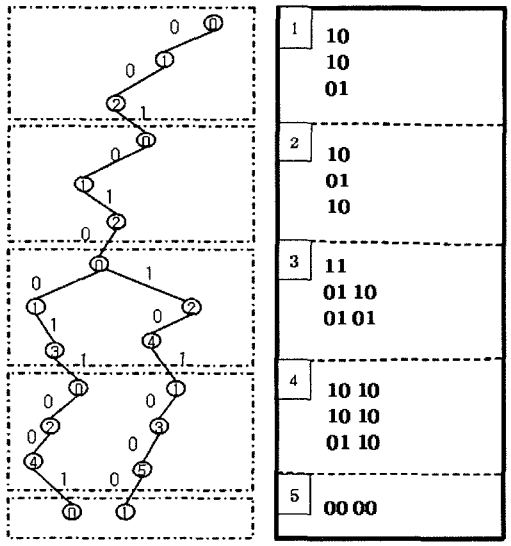

그림 5 이진 트라이에 S의 두 번째 서브 시퀀스 'ACT$' 삽입

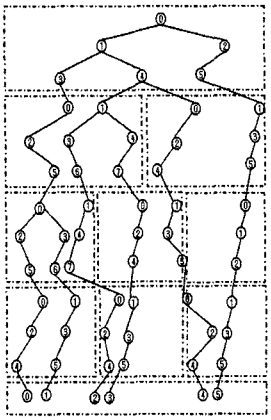

그림 6 이진 트라이의 구성 예

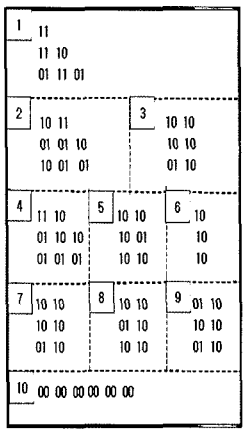

그림 7 이진 트라이의 내부 표현 예

#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 8 페이지 테이블

Offset
0
3
1
4
2
5

그림 9 단말 테이블

그림 4, 5, 6, 7의 예에서 보이는 점선 영역은 디스크 상의 페이지 분할 영역을 나타낸다. 또한 그림 4, 5, 6에서 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 4의 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 6과 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 7의 인덱스 구조의 페이지 교체를 위해서는 그림 8과 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 9에 그 예를 보인다.

4. 질의처리 방안

본 장에서는 제 3장에서 제안한 트라이 인덱스를 이용한 유사 서브 시퀀스 검색 기법에 대하여 논의한다. 제 4.1절에서는 기본적인 트라이 인덱스 검색 방식을 보인다. 제안된 방식은 트라이 인덱스를 너비 우선 방식으로 운행하여 경로 상에 존재하는 모든 유사 서브 시퀀스를 효율적으로 검색한다. 다음 제 4.2절에서는 질의 길이가 윈도우의 크기보다 큰 일반적인 경우(|Q|>>|W|)에 대한 분할 질의 처리 알고리즘을 보이고, 이를 이용한 질의 처리 최적화 방안을 보인다.

4.1 트라이 인덱스 검색 기법

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[7,9].

접미어 트리에서는 경로 운행에 의하여 모든 서브 시퀀스를 발견할 수 있다. 접미어 트리 기반의 유사 검색 알고리즘에서는 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[9,11,13,16]. 이진 트라이를 이용한 유사 검색을 위하여 접미어 트리 기반의 기존 방식을 그대로 적용하여 사용할 수 있다. 그러나 제안된 트라이 인덱스는 2비트의 노드 정보만을 저장할 뿐 노드와 노드를 연결하는 포인터 정보, 노드 레벨 정보, 경로 상의 해당 서브 시퀀스 정보를 저장하지 않는다. 따라서 인덱스 검색을 위하여는 해당 페이지의 노드 정보를 순차적으로 읽어 이 들의 정보를 얻어야 한다. 또한 이진 트라이 인덱스는 동일 노드 레벨의 노드 정보가 같은 페이지에 연속적으로 배열되는 특성을 갖는 디스크 기반의 인덱스 구조이다. 따라서 깊이 우선 방식에 의하여 트라이 상의 모든 경로를 검색하는 경우, 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 2개의 큐(queue) 구조를 이용한다. Qpagenumber는 탐색 대상의 페이지 정보를 담고 있으며, Qnode는 각 페이지의 탐색 대상 노드 정보를 담고 있다. 우선, 트라이 인덱스의 루트 페이지 번호를 Qpagenumber에, 루트 페이지에 대한 Qnode에 루트 노드를 각각 푸시한다(Lines 1~2). 전체 알고리즘은 페이지와 노드 탐색을 위한 이중의 while 문으로 이루어져 있다. 외부 while 문(Lines 4~24)의 Line 5는 탐색 대상의 페이지 번호를 팝업하는 과정을 나타내며, 내부 while 문(Lines 7~24)의 Line 8에서는 탐색 대상의 노드 번호를 팝업하는 과정을 나타낸다.

Algorithm 2: Search-Trie

Input : binary trie I, query sequence Q, tolerance T, page table P, leaf table L, maxNode M
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.
7.      while (notEmpty(Qnode[pageNumber])) do
8.          current_Node = pop(Qnode[pageNumber]);
9.
10.         for each child node CNi of the current_Node do
11.             moreVisit = TRUE;
12.             AppedBitString(CNi_Path, current_Node, CNi);
13.
14.             if BitCount(CNi_Path) mod 3 == 0 then
15.                 DPT_CNi = AddColumn(current_DPT, CNi_Path),
16.                 Let dist be the last row value of the new added column;
17.                 if dist <= T then
18.                     FindAnswer(CNi, L);
19.                     moreVisit = FALSE;
20.             else moreVisit = FurtherVisit(DPT_CNi);
21.             if moreVisit
22.                 if terminal_Node(CNi) then FindCandidateAnswer(CNi, L);
23.                 else CheckpageAndPush(Qpagenumber, Qnode, CNi, P, M);
24.
```

현재의 탐색 노드 current_Node의 모든 차일드 노드 CN_i에 대하여 다음과 같은 검색을 수행한다(Lines 10~24). 우선 moreVisit는 이 후의 노드 검색을 계속 수행할지의 여부를 결정하는 변수로서 초기 값을 TRUE로 설정한다(Line 11). AppendBitString()은 CN_i의 방문에 의하여 경로 정보 CN_i_Path를 생성하는 함수를 나타낸다(Line 12). 이렇게 생성된 경로 정보 CN_i_Path의 길이가 3이라면, 3비트의 문자로 이루어진 경로가 방문되었음을 의미하므로 DP 테이블에 생성된 경로에 대한 새로운 열을 추가한다. 함수 AddColumn()이 이일을 수행하며, current_Node까지 생성된 DP 테이블 current_DPT를 사용하여 새로운 열이 추가된 DPT_CN_i를 생성하는 과정을 나타낸다(Line 15). 여기에서 DPT_CN_i 테이블의 새로운 열의 마지막 요소 값을 dist로 한다(Line 16). 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(Line 19). 그러나 dist의 값이 T 보다 큰 경우에는 함수 FurtherVisit()에 의하여 이 경로에 대한 노드 탐색을 계속할 것인지, 중지할 것인지의 여부를 결정한다. 다음, Lines 21~23의 과정은 moreVisit의 값이 TRUE인 경우의 경로 탐색 과정을 나타낸다. 만약 CN_i가 단말 노드인 경우에는 함수 FindCandidateAnswer()를 호출하여 해당 노드 CN_i를 후보 결과로 등록하고, 후처리 과정에 의하여 유사 서브 시퀀스를 검색한다(Line 22). 이 과정은 질의 길이가 윈도우 서브 시퀀스의 길이보다 큰 경우의 질의 처리를 위하여 필요하다. 그 외의 경우에는 함수 CheckpageAndPush()를 이용하여 CN_i를 Qnode에 푸시하고, 다음 노드 검색을 수행한다(Line 23). 단, 현재 처리 중인 노드의 개수가 Trie 인덱스 페이지의 최대 노드 수 maxNode를 초과하는 경우에는 페이지 테이블 P를 참조하여 탐색할 새로운 페이지를 Qpage-number에 푸시하고, 해당 Qnode에 CN_i가 푸시될 수 있도록 한다.

위에서 설명한 바와 같이 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. Line 18의 FindAnswer() 함수가 이 역할을 담당한다. 이 과정은 일반적으로 상당한 오버헤드를 필요로 할 수 있다. 그러나 제안된 트라이 인덱스에서는 서브 트라이의 모든 중간 노드를 검색하지 않고 다음과 같은 간단한 과정에 의하여 모든 단말 노드를 검색할 수 있다.

해당 노드 CN_i의 서브 트라이에서 해당 노드 CN_i 이후의 최좌측 노드만을 방문하여 얻어진 단말 노드를 LCN_i이라 하고, 최우측 노드들만을 방문하여 얻어진 단말 노드를 RCN_i라고 한다. 트라이 인덱스는 동일 길이의 윈도우 서브 시퀀스를 오름차순의 정렬된 순서로 저장하고 있으므로 모든 단말 노드는 트라이 구조의 동일 노드 레벨에 위치하며, 각 인덱스 페이지 내에 삽입된 순서에 따라 연속으로 저장된다. 따라서 노드 CN_i의 서브 트라이에 존재하는 단말 노드는 LCN_i로부터 RCN_i 사이에 존재하는 노드에 해당한다. 즉 LCN_i와 RCN_i를 검색함으로써 그 사이에 존재하는 모든 단말 노드를 직접 검색할 수 있다.

직관적인 이해를 위하여 제 3장에서 예제로 사용한 트라이 인덱스를 이용하여 질의 시퀀스 'AGC'와 에디트 거리 T가 1 이하인 유사 서브 시퀀스를 검색하는 과정의 일부분을 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)', '$(000)'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

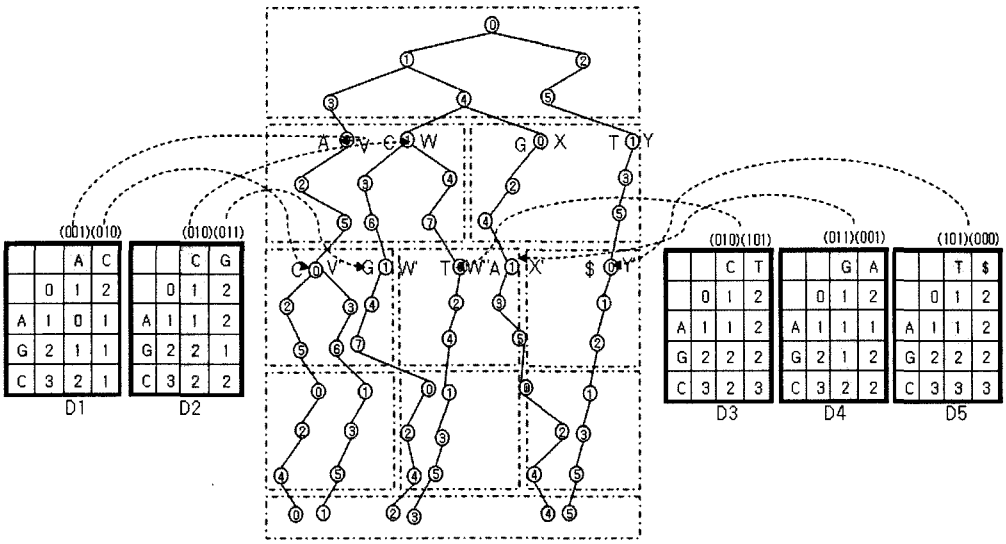

그림 10 너비 우선 방식의 이진 트라이 운행 예

페이지는 최대 한번만 액세스 되는 효율적인 검색이 가능함을 알 수 있다.

4.2 유사 서브 시퀀스 검색 기법

이진 트라이 인덱스는 일정 길이의 윈도우 서브 시퀀스를 인덱싱 대상으로 하고 있으므로 |Q| >> |W|의 경우, Algorithm 2에서 보인 함수 FindCandidateAnswer()에 의한 후보 결과 집합이 커져 후처리를 위한 시간이 증가하게 되며 따라서 전체 질의 처리 시간이 증가하게 된다. 본 연구에서는 질의 처리 성능을 최적화하기 위하여 질의 길이와 트라이 인덱스의 윈도우 크기를 고려하여 질의를 적절히 분할하여 처리하는 분할 질의 처리 기법[16]을 사용한다.

질의 Q와 유사 허용치 T가 주어진 경우, 분할 질의 처리 방식에 의한 질의 처리 과정은 다음과 같다.

(1) 질의 Q를 p개의 서브 질의 SQ_1, SQ_2, ..., SQ_p로 분할한다. p의 개수는 |Q|와 |W|의 크기에 따라 최적의 값으로 결정한다. 단, 여기에서 서브 질의의 크기는 동일할 필요는 없다.

(2) 각 서브 질의 SQ_i (i = 1, 2, ..., p)에 대하여 트라이 인덱스를 이용한 유사 서브 시퀀스 검색을 수행한다. 이 때 각 서브 질의를 위한 유사 허용치는 $\lfloor T/p \rfloor$로 주어진다.

(3) (2)의 과정에 의하여 얻어진 결과 집합에 대하여 후처리를 수행한다. 후처리 과정에서는 각 시퀀스 오프셋 i에 대하여 하여 다이나믹 프로그래밍 방식에 의한 유사 서브 시퀀스 검색을 수행하며, 이 때 질의 Q와 유사 허용치 T가 사용된다.

이를 위한 유사 서브 시퀀스 검색 알고리즘을 Algorithm 3에 보인다. 동작 과정을 간단히 설명하면 다음과 같다. 함수 Search-Trie-By-SubQuery()에서는 질의 Q를 partitionQuery()를 이용하여 적절한 길이와 유사 허용치를 갖는 p개의 서브 질의로 분할한다(Line 1). 이 때 서브 시퀀스의 분할 개수 및 길이는 질의 길이 변화에 따른 함수 Search-Trie()의 성능을 고려하여 결정한

Algorithm 3: Search-Trie-By-SubQuery

```
Input : binary trie I, query sequence Q, tolerance T, page table P, leaf table L,
        maxNode M
Output : set of answers answerSet
1. p = partitionQuery(Q, T);
2. for each subquery SQi do
3.     candidateSet = candidateSet ∪ Search-Trie(I, SQi, ⌊T/p⌋, P, L, M);
4. answerSet = postProcessing(candidateSet, Q, T);
5. return answerSet;
```

다. 다음, 분할하여 생성된 각 서브 질의 SQ_i에 대하여 Algorithm 2의 함수 Search-Trie()를 호출하여 유사 서브 시퀀스 검색을 수행한다(Lines 2~3). 이때 주의할 점은 각 서브 질의를 위한 유사 허용치가 $\lfloor T/p \rfloor$로 동일하게 주어진다는 점이다. 다음, 함수 postProcessing()는 후보 결과 집합 candidateSet에 대하여 후처리를 수행하여 최종 결과 집합을 생성한다(Line 4). 후처리 과정은 각 후보 결과 i에 대하여 DNA 시퀀스 S[i-|Q|-T, ..., i+|Q|+T]와 질의 Q에 대한 다이나믹 프로그래밍 방식에 의한 유사 서브 시퀀스 검색 과정을 나타낸다.

Search-Trie-By-SubQuery의 질의 처리 시간은 $p*$ Search-Trie() + $p*C_i*|Q|^2$로 주어진다. 여기에서 $p*$ Search-Trie()는 인덱스 검색 시간을 나타내며, $p*C_i*|Q|^2$는 후처리 시간을 의미한다. p는 분할 질의 개수를 나타내며, Search-Trie()는 서브 질의 SQ_i에 대한 유사 허용치 $\lfloor T/p \rfloor$의 트라이 인덱스 검색 시간을 의미한다. 또한 C_i는 서브 질의 SQ_i에 의한 후보 결과의 개수를 나타내며, $|Q|^2$는 각 후보 결과에 대한 다이나믹 프로그래밍 기법에 의한 후처리 시간을 의미한다.

5. 성능 평가

본 장에서는 실험에 의한 성능 분석을 통하여 제안하는 기법의 우수성을 규명한다. 본 실험에서는 GenBank [28]로부터 다운 받은 Human Chromosome 19, 21 시퀀스를 이용한다. 실험에 사용된 이 들 DNA 시퀀스에는 기본적으로 Σ = {A, C, G, T}의 네 종류 문자가 출현하며, 출현 빈도가 높지 않은 N, S, Y 등의 와일드 카드 문자가 출현한다. 실험에 사용된 각 DNA 시퀀스에는 최대 7개의 서로 다른 문자가 출현하고 있다. 따라서 본 실험에서는 '$' 문자를 포함하여 8가지 문자를 처리 대상으로 한다. 실험을 위한 하드웨어 플랫폼으로는 Redhat Linux 9(Kernel Version 2.4.20)를 운영체제로 사용하고, 1GB의 주기억 장치, 80GB 디스크를 갖는 Pentium IV 3.2GHz의 PC를 사용한다.

제안된 기법의 성능을 평가하기 위하여 인덱스의 크기 및 유사 서브 시퀀스 검색을 위한 질의 처리 시간을 기존 방식과 실험을 통하여 비교 분석한다. Search-Trie는 본 논문에서 제안한 트라이 인덱스 검색 방식을 나타낸다. 성능 비교를 위하여 접미어 트리를 이용한 유사 서브 시퀀스 검색 방식(Suffix)와 DP 기법에 근거한 Smith-Waterman 방식(SW)을 사용한다. Suffix는 Top-Down Disk-based 접미어 트리 알고리즘[22]을 사용한 오픈 소스 프로그램(http://www.eecs.umich.edu/tdd)를 사용하였다. 실험 데이타로는 28.6Mbp의 Human Chromosome 21 시퀀스(Chr 21) 일부와 56Mbp의 Human Chromosome 19 시퀀스(Chr 19) 일부를 사용하였다. 질의 시퀀스는 실험 데이타 셋으로부터 랜덤 방식에 의하여 추출한 것을 사용하였으며, 각 실험은 100번의 반복 실험 결과를 평균하여 나타낸다.

먼저 실험 1에서는 본 논문에서 제안한 Search-Trie의 인덱스 크기를 Suffix와 비교, 평가한다. 표 1에 데이타 크기의 변화에 따른 인덱스의 크기 변화를 비교한 결과를 보인다. 본 실험에서는 Search-Trie 인덱스 구성을 위하여 윈도우 크기를 15로 설정하였다. 페이지 크기로 4K를 설정하였다. Suffix 인덱스는 내부 노드와 단말 노드를 표현하는 1차원 배열의 Tree 인덱스와 Leaf_Table, Rmost_Table로 구성된다. Rmost_Table은 Tree 인덱스 내의 각 노드들의 Level 값을 계산하기 위한 정보를 담고 있으며, Leaf_Table은 각 노드의 단말 여부를 알려주는 기능을 수행한다. Search-Trie 인덱스는 이진 트라이 인덱스 Binary_Trie와 페이지 테이블 Page_Table, 단말 테이블 Leaf_Tabl로 구성된다. 표 1의 비교 결과로부터 Suffix, Search-Trie의 두 방식 모두 데이타의 크기 변화에 따라 거의 선형적으로 인덱스 크기가 비례하여 증가함을 알 수 있다. 그러나 절대 크기를 비교 할 경우 Search-Trie는 Suffix에 비하여 약 40%의 저장 공간만을 필요로 한다.

다음, 실험 2에서는 Search-Trie의 질의 처리 시간을 Suffix와 비교한다. Search-Trie는 |W|=15의 Trie 인덱스를 사용한다. 질의 처리 시간은 DNA 시퀀스 S 상에서 주어진 질의 시퀀스 Q와 에디트 거리 T 이하의 유사 허용치를 만족하는 모든 유사 서브 시퀀스 S'를 검색하여 그 오프셋 값을 반환하는 총 시간을 의미한다. 또한 Search-Trie와 Suffix의 실행 결과 얻어지는 유사 서브 시퀀스의 검색 개수를 측정한다.

우선, 질의 길이 변화에 따른 Search-Trie의 평균 질

표 1 인덱스의 크기 비교

Data Size	Suffix				Search-Trie			
	Tree	Leaf_ Table	RMost_ Table	Total	Binary_ Trie	Leaf_ Table	Page_ Table	Total
28.6Mbp (Chr 21)	267.6M	46.4M	46.4M	360.4M	50.2M	114.4M	0.5M	165.1M
56Mbp (Chr 19)	539M	91M	91M	721M	71.9M	224.1M	0.7M	296.7M

의 처리 시간을 Suffix와 비교한다. 그림 11에 28.6Mbp와 56Mbp의 DNA 시퀀스를 위한 질의 처리 시간을 각각 보인다. 여기에서 유사 허용치 T의 값으로 각 질의 길이의 10%에 해당하는 값을 사용하였다. 이 결과로부터 Search-Trie는 질의 길이가 30이하의 경우, Suffix에 비하여 약 4배에서 9배의 고속 검색 결과를 보이고 있다. 그러나 Search-Trie는 질의 길이 40 이상의 질의 처리에 대하여 Suffix에 비하여 질의 처리 시간이 급격히 증가한다. 그 이유는 Trie 인덱스가 |W|=15의 윈도우 서브 시퀀스를 사용하므로 |Q|>>|W|의 경우 검색 결과 중 후보 결과가 증가하여 이에 따른 후처리 시간이 증가하기 때문이다.

다음, 유사 허용치의 변화에 따른 Search-Trie의 평균 질의 처리 시간을 Suffix와 비교한다. 그림 12에 28.6Mbp와 56Mbp의 DNA 시퀀스를 위한 질의 처리 시간을 각각 보인다. 여기에서는 질의 길이로서 |Q|=30을 사용하였다. 실험 결과로부터 Search-Trie는 T의 값이 3이하의 경우, Suffix에 비하여 약 4배에서 17배의 고속 검색 결과를 보임을 알 수 있다. 그러나 Search-Trie는 T의 값이 증가하면 Suffix에 비하여 질의 처리 시간이 증가하며, 그 이유는 그림 11의 경우와 같이 후보 결과의 증가에 따른 후처리 시간의 증가를 그 원인으로 들 수 있다.

한편, 그림 11과 그림 12의 결과로부터 Suffix는 질의 길이 및 유사 허용치 증가에 따라 실행 시간이 거의 선형적으로 증가하고 있는 것으로 보인다. 그러나 Suffix는 깊이 우선 방식에 의한 인덱스 검색을 위하여 다량의 인덱스 페이지를 동시에 주기억 장치에 반복적으로 적재할 필요가 있으므로 질의 길이 혹은 유사 허용치가 증가하면 인덱스 탐색 공간이 급격히 증가하게 되어 프로그램 실행이 불가능해진다. 실제로 우리의 실험에서는 56Mbp의 DNA 시퀀스를 실험 대상으로 하여 |Q|=60 이상의 경우(유사 허용치 10%) 혹은 T=6 이상의 경우(|Q|=30), 그 결과를 얻을 수 없었다.

다음, 실험 3에서는 Search-Trie-By-SubQuery와 Search-Suffix-By-SubQuery의 질의 처리 시간을 비교한다. Search-Trie-By-SubQuery는 Search-Trie 기법과 최적의 p 값을 사용한 분할 질의 처리 기법을 기반으로 하는 우리의 유사 서브 시퀀스 검색 기법을 나타낸다. 또한 Search-Suffix-By-SubQuery는 Search-Trie를 이용하는 대신에 Suffix를 사용하는 분할 처리 기법을 나타낸다. 이 때 Search-Trie-By-SubQuery와 Search-Suffix-By-SubQuery를 위한 서브 질의의 개수 p의 값은 실험 2에서 보인 바와 같은 질의 크기 및 유사 허용치 변화에 따른 인덱스의 성능과 후처리 오버헤드를 고려하여 최적의 값으로 결정한다.

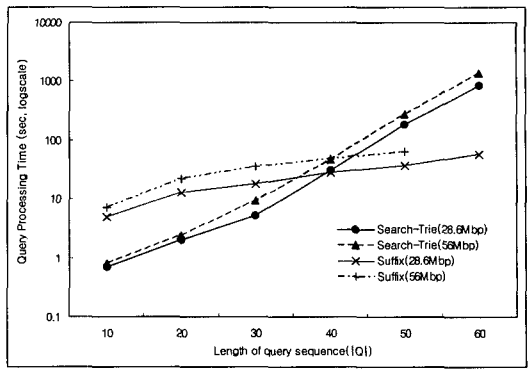

그림 11 질의 길이 및 데이타 사이즈 변화에 따른 질의 처리 시간 비교

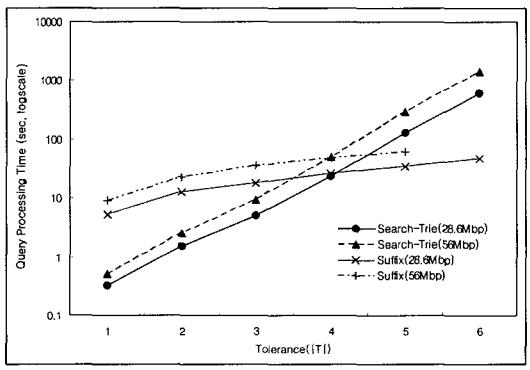

그림 12 유사 허용치 및 데이타 사이즈 변화에 따른 질의 처리 시간 비교

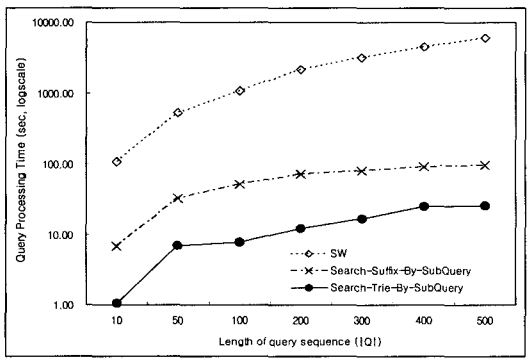

그림 13 질의 길이 변화에 따른 질의 처리 시간 비교

그림 13에 질의 길이의 증가에 평균 질의 처리 시간을 보인다. 여기에서 Y축은 질의 처리 시간을 나타내며 로그 스케일링 되어 있다. 실험 데이타로서 56Mbp의 DNA 시퀀스를 사용하였으며, 유사 허용치 T의 값은 각 질의 길이의 10%에 해당하는 값을 사용하였다. Search-Trie-By-SubQuery에서는 서브 질의의 길이를

25로 균등 분할하였으며, Search-Suffix- By-SubQuery의 경우, 질의 길이에 따라 40과 20의 서브 질의를 혼합 사용하여 분할 질의 처리를 수행하였다.

이들 실험 결과로부터 제안된 기법은 대규모 DNA 시퀀스를 대상으로 하여 타 방식에 비하여 우수한 성능을 보이며, 실시간 내에 효율적인 유사 서브 시퀀스 검색 결과를 얻을 수 있음을 알 수 있다. Search-Trie-By-SubQuery는 Search-Suffix-By-SubQuery와 SW에 비하여 각각 3배~5배와 75배~200배의 성능 개선 효과를 보인다.

6. 결 론

본 논문에서는 효율적인 DNA 시퀀스 검색을 위한 인덱스 구조와 유사 검색 알고리즘을 제안하였다. 제안된 인덱스는 포인터 없이 트라이를 비트 스트링으로 표현하는 이진 트라이를 기본 구조로 하며, DNA 시퀀스를 구성하는 모든 문자의 시작 위치로부터 윈도우 서브 시퀀스를 추출하여 인덱싱 대상으로 한다. 이러한 구조의 장점은 저장 공간의 오버헤드를 크게 줄일 수 있다는 점과 전체 구조를 페이지 기반으로 형성시킬 수 있다는 점, 고속의 단말 노드 검색이 가능하다는 점을 들 수 있다. 따라서 제안된 인덱스 구조를 사용함으로써 기존의 접미어 트리가 가지던 저장 공간, 검색 성능, DBMS와의 통합 등의 문제점들을 모두 해결할 수 있다. 제안된 인덱스를 기반으로 하는 유사 서브 시퀀스 검색 알고리즘은 너비 우선 방식으로 인덱스 경로를 탐색하며, 다이나믹 프로그래밍 기법을 사용하여 이진 트라이 경로 상에 존재하는 모든 유사 서브 시퀀스를 효율적으로 검색한다. 특히 질의 길이가 긴 경우에는 질의 분해에 의한 서브 질의 검색 방식을 사용하여 정확도의 손실 없이 유사한 모든 서브 시퀀스를 효율적으로 검색해낸다.

제안된 기법의 우수성을 검증하기 위하여, 실험을 통한 성능 평가를 수행하였다. 실험 결과에 의하면, 제안된 인덱스는 기존의 접미어 트리와 비교하여 더 작은 저장 공간을 가지고도 기존의 접미어 트리에 비하여 수배에서 수십배까지의 검색 성능의 개선 효과를 나타내며, 특히 질의 길이가 길거나 유사 허용치가 큰 경우에도 유사 검색 알고리즘은 효율적으로 동작하여 Suffix과 SW에 비하여 각각 수배에서 수십배까지의 검색 성능의 개선 효과를 나타내는 것으로 나타났다.

참 고 문 헌

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

[2] Z. Tan, X. Cao, B. Ooi, and A. Tung, "The ed-tree: An Index for Large DNA Sequence Databases," In Proceedings of SSDBM Conference, pp. 1-10, 2003.

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

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

[5] 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.

[6] 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.

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

[8] 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.

[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] 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.

[11] 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.

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

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

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

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

[16] 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.

[17] H. Wang et al., "BLAST++: A Tool for BLASTing Queries in Batches," In Proceedings First Asia-Pacific Bioinformatics Conference, pp. 71-79, 2003.
[18] E. Horowitz, S. Sahni, and S. Anderson-Freed, Fundamentals of Data Structures in C, Computer Science Press, 1993.
[19] 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.
[20] 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.
[21] T. Kahveci and A. K. Singh, "An Efficient Index Structure for String Databases," In Proceedings of the 27th VLDB Conference, pp. 351-360, 2001.
[22] S. Tata, R. Hankins, and J. Patel, "Practical Suffix Tree Construction," In Proceedings of the 30th VLDB Conference, pp. 36-47, 2004.
[23] K. Kelly and P. Labute, "The A* Search and Applications to Sequence Alignment," http://www.chemcomp.com/article/astar.htm, 1996.
[24] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," SIAM J. Comput. 22, pp. 935-948, 1993.
[25] V. Makinen, "Compact Suffix Array: A Space efficient Full-text Index," Fundamenta Informaticae, 56(1-2), pp. 191-210, 2003.
[26] V. Makinen and G. Navarro, "Compressed Compact Suffix Arrays," CPM 2004, Springer-Verlag LNCS 3109, pp. 420-433.
[27] 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.
[28] http://www.ncbi.nlm.nih.gov

원 정 임

1992년 2월 한림대학교 전자계산학과 졸업(학사). 1997년 8월 한림대학교 전자계산학과 졸업(석사). 2004년 2월 한림대학교 전자계산학과 졸업(박사). 2000년 3월~2004년 2월 한림대학교 교양교육부 강의전담교수. 2004년 3월~2006년 2월 연세대학교 컴퓨터과학과 연구교수. 2006년 3월~6월 서울대학교 유전자이식연구소 선임연구원. 2006년 7월~현재 한양대학교 정보통신대학 정보통신학부 연구교수. 관심분야는 데이타베이스 시스템, 데이타 마이닝, XML 응용, 바이오정보공학, 데이타베이스 보안, 이동객체 데이타베이스, 텔레매틱스

홍 상 균

2005년 2월 한림대학교 정보통신공학부 졸업(학사). 2007년 2월 한림대학교 정보통신공학부 졸업(석사). 관심분야는 XML, 데이타베이스 시스템, 바이오인포매틱스, 저장시스템

윤 지 희

1982년 2월 한양대학교 전자공학과 졸업(학사). 1985년 3월 일본 구주대학교 정보공학과 졸업(석사). 1988년 3월 일본 구주대학교 정보공학과 졸업(박사). 1998년 3월~1999년 2월 미국 UCLA대학교 전산학과 방문교수. 1988년 4월~현재 한림대학교 정보통신공학부 교수. 관심분야는 바이오인포매틱스, 데이타 마이닝, XML, 공간 데이타베이스/GIS

박 상 현

정보과학회논문지 : 데이타베이스 제 34 권 제 1 호 참조

김 상 욱

1989년 2월 서울대학교 컴퓨터공학과 졸업(학사). 1991년 2월 한국과학기술원 전산학과 졸업(석사). 1994년 2월 한국과학기술원 전산학과 졸업(박사). 1991년 7월~8월 미국 Stanford University, Computer Science Department 방문 연구원 1994년 2월~1995년 2월 KAIST 정보전자연구소 전문 연구원. 1999년 8월~2000년 8월 미국 IBM T.J. Watson Research Center Post-Doc. 1995년 3월~2000년 8월 강원대학교 컴퓨터정보통신공학부 부교수. 2003년 3월~현재 한양대학교 정보통신대학 정보통신학부 교수. 관심분야는 데이타베이스 시스템, 저장 시스템, 트랜잭션 관리, 데이타 마이닝, 멀티미디어 정보 검색, 공간 데이타베이스/GIS, 주기억장치 데이타베이스, 이동 객체 데이타베이스/텔레매틱스, 사회 연결망 분석, 웹 데이타 분석