대용량 DNA 시퀀스 데이타베이스를 위한 효율적인 인덱싱

(Efficient Indexing for Large DNA Sequence Databases)

원 정 임 † 윤 지 희 †† 박 상 현 ††† 김 상 욱 ††††
(Jung-Im Won) (Jee-Hee Yoon) (Sang-Hyun Park) (Sang-Wook Kim)

요 약 DNA 시퀀스 검색은 분자 생물학 분야에서 사용되는 매우 중요한 연산이다. DNA 시퀀스 데이타베이스는 매우 큰 용량을 가지므로 DNA 시퀀스 검색의 효율적인 처리를 위해서는 고속 인덱스의 사용이 필수적이다. 본 논문에서는 DNA 시퀀스 검색을 위하여 기존에 제안된 접미어 트리가 가지는 저장 공간, 검색 성능, DBMS와의 통합 등의 문제점들을 지적하고, 이러한 문제점을 해결할 수 있는 새로운 인덱스를 제안한다. 제안된 인덱스는 포인터 없이 트라이를 비트 스트링으로 표현하는 기본 구조와 후처리 시 액세스 되어야 하는 트라이의 단말 노드를 신속하게 찾기 위한 보조 자료 구조로 구성된다. 또한, 제안된 인덱스를 이용하여 DNA 시퀀스 검색을 효과적으로 처리하는 알고리즘을 제시한다. 제안된 기법의 우수성을 검증하기 위하여, 실험을 통한 성능 평가를 수행하였다. 실험 결과에 의하면, 제안된 인덱스는 기존의 접미어 트리와 비교하여 더 작은 저장 공간을 가지고도 13배에서 29배까지의 검색 성능의 개선 효과를 가지는 것으로 나타났다.

키워드 : DNA 시퀀스 데이타베이스, 인덱싱, 트라이

Abstract In molecular biology, DNA sequence searching is one of the most crucial operations. Since DNA databases contain a huge volume of sequences, a fast indexing mechanism is essential for efficient processing of DNA sequence searches. In this paper, we first identify the problems of the suffix tree in aspects of the storage overhead, search performance, and integration with DBMSs. Then, we propose a new index structure that solves those problems. The proposed index consists of two parts: the primary part represents the *trie* as bit strings without any pointers, and the secondary part helps fast accesses of the leaf nodes of the trie that need to be accessed for post processing. We also suggest an efficient algorithm based on that index for DNA sequence searching. To verify the superiority of the proposed approach, we conducted a performance evaluation via a series of experiments. The results revealed that the proposed approach, which requires smaller storage space, achieves 13 to 29 times performance improvement over the suffix tree.

Key words : DNA sequence database, indexing, trie

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 시퀀스 검색 연산은 이미 특성이 판명된 DNA 시퀀스 데이타베이스로부터 주어진 DNA 시퀀스와 염기 배열이 유사한 시퀀스들을 검색하는 연산이다. 분자 생물학자들은 이러한 DNA 시퀀스 검색 연산을 통하여 새롭게 발견된 DNA 시퀀스의 역할, 진화 과정, 화학적 구조 등을 유추한다. 최근의 연구 보고[2]에 의하면 분자 생물학자들이 일상적으로 수행하는 작업 중 35.2%가 시퀀스 검색 연산에

· 본 연구는 한국과학재단 목적기초연구(R04-2003-000-10048-0)와 2003년도 연세대학교 신진 교수 지원 과제(2003-1-0361), 정보통신부 2001년 대학기초연구의 지원으로 수행되었습니다.
† 정 회 원 : 연세대학교 컴퓨터과학과 교수
jiwon@cs.yonsei.ac.kr
†† 종신회원 : 한림대학교 정보통신공학부
jhyoon@hallym.ac.kr
††† 종신회원 : 연세대학교 컴퓨터과학과 교수
sanghyun@cs.yonsei.ac.kr
†††† 종신회원 : 한양대학교 정보통신학부
wook@hanyang.ac.kr
논문접수 : 2004년 1월 27일
심사완료 : 2004년 6월 25일

할애되며, 그 중 33.3%는 DNA 시퀀스 검색에 해당하는 것으로 보고되고 있다.

DNA 시퀀스 데이타베이스는 매우 큰 용량을 가진다. DNA 시퀀스 데이타베이스의 하나인 GenBank[3]는 초기인 1992년 약 1억의 염기들로 구성된 7만 8천여 개의 DNA 시퀀스들을 가졌으나, 2003년 말에는 약 365억 염기들로 구성된 3천만여 개의 DNA 시퀀스들을 가지는 거대한 규모로 성장하였다. 또한, GenBank는 현재도 약 14개월마다 그 규모가 두 배로 늘어나고 있으며, 이러한 증가율 또한 지속적으로 증가하고 있다.

DNA 시퀀스 검색 문제는 일반적으로 다음과 같이 정의된다[4,5]. DNA 시퀀스 데이타베이스 S와 질의 시퀀스 Q, 유사 허용치 T가 주어지면, Q의 서브 시퀀스 Q′과 유사 허용치가 T 이상인 서브 시퀀스 S′을 S 내에서 검색해 낸다. 이때 Q′과 S′ 사이의 유사도를 계산하기 위하여 응용 목적에 적합한 유사도 함수(similarity function)를 사용한다. DNA 시퀀스 검색을 위하여 일반적으로 사용되는 방식은 데이타베이스 전체를 검색 대상으로 하는 순차 검색 기법이다[6-8]. BLAST[6,7]는 DNA 시퀀스 검색 연산을 위하여 가장 널리 사용되는 표준 도구이다. BLAST는 매치시키고자 하는 두 시퀀스 S와 Q에 대하여 먼저 시드(seed)라 불리는 짧은 고정 길이의 워드(word) 쌍을 S와 Q로부터 찾은 후, 이 시드로부터 유사한 부분을 점차 확장하는 방식을 사용한다. 또한 최근에는 BLAST의 성능 향상을 위한 다양한 방식들[9,10]이 제안되고 있다. 단, 이 들 기법은 데이타베이스의 순차 검색을 기본으로 하기 때문에 데이타베이스 크기가 커지는 경우 검색 효율이 저하하는 단점을 갖는다.

최근의 DNA 시퀀스 데이타베이스 규모의 급격한 증가 추세를 고려할 때, DNA 시퀀스 검색 연산을 보다 효과적으로 지원하기 위한 인덱싱 기술이 요구된다. 접미어 트리(suffix tree)[11]는 DNA 시퀀스 검색을 위한 좋은 인덱스 구조로 알려져 왔다[12,13]. 접미어 트리는 주어진 시퀀스의 접미어에 해당되는 서브시퀀스들을 트리 형태로 구성한 것으로서 서브시퀀스 검색의 효율이 우수하며, 유사 검색을 위한 알고리즘들이 이미 고안되어 있다는 장점을 갖는다[14-16]. 또한, 기존에는 디스크 상에 주기억장치 용량 이상인 접미어 트리를 구성하는데 어려움이 있었으나, 참고 문헌 [17]에서 이 문제를 해결한 바 있다. 그러나 접미어 트리는 그 구조적 특성으로 인하여 다음과 같은 문제점을 갖는다.

① 저장 공간: 참고 문헌 [17]에 의하면, 286M 염기를 가지는 DNA 시퀀스를 대상으로 구성된 접미어 트리의 크기는 19G 바이트로 나타났다. 따라서 접미어 트리는 저장 공간의 오버헤드가 매우 크다.

② 검색 성능: 접미어 트리의 크기가 큰 만큼 이 트리를 순회하는 시간이 길다. 따라서 전체 검색 성능이 떨어진다.

③ DBMS와의 통합: 접미어 트리는 그 구조적 특성 상 페이지 단위로 구현하기가 어렵다. 따라서 모든 데이타를 페이지 단위로 구현하는 DBMS와 밀 결합(seamless integration)에 어려움이 있다[11,18].

본 논문에서는 이와 같은 접미어 트리의 문제점들을 해결하는 DNA 시퀀스 검색을 위한 새로운 인덱스를 제안한다. 제안된 인덱스는 트라이(trie)[11,19]를 기본 구조로 채택하며, 포인터 없이 트라이를 비트 스트링으로 표현하는 방식을 채택한다. 또한, DNA 시퀀스 검색을 위한 이러한 트라이의 순회 시 검색 대상이 되는 트라이의 단말 노드를 빠르게 찾을 수 있는 보조 자료 구조로서 다차원 인덱스 구조를 사용한다. 이러한 특성으로 인하여 제안된 인덱스는 기존의 접미어 트리가 가지던 저장 공간, 검색 성능, DBMS와의 통합 등의 문제점들을 모두 해결할 수 있다. 또한, 제안된 인덱스를 이용하여 DNA 시퀀스 검색을 효과적으로 처리하는 질의 처리 알고리즘을 제시한다. 다양한 실험을 통하여 제안된 인덱스 구조를 이용한 검색 성능을 기존의 방식과 비교하여 정량적으로 검증한다. 실험 결과에 의하면, 제안된 인덱스는 기존의 접미어 트리와 비교하여 더 작은 저장 공간을 가지고도 13배에서 29배까지의 검색 성능의 개선 효과를 가지는 것으로 나타났다.

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 DNA 시퀀스 검색의 처리 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 관련 연구 및 배경 기술에 대하여 요약한다. 먼저, 제 2.1절에서는 DNA 시퀀스 검색을 위한 관련 연구 결과를 순차 검색 기반과 인덱스 기반 검색으로 나누어 요약, 설명한다. 제 2.2절에서는 DNA 시퀀스의 인덱싱에 적합하다고 알려진 접미어 트리에 관하여 설명한다.

2.1 기존 연구

순차적 검색을 기반으로 하는 시퀀스 검색 기법으로 Smith-Waterman(S-W) 알고리즘[8]을 들 수 있다. S-W 알고리즘은 다이나믹 프로그래밍(dynamic programming)기법에 의하여 두 시퀀스 S와 Q 사이의 최적 정렬(optimal alignment)을 구하는 대표적인 알고리

즘으로 알려져 있다. 그러나 이 방식은 두 시퀀스의 길이의 곱에 비례하는 계산량(O(|Q|×|S|))를 필요로 하여 속도가 느린 단점이 있다.

BLAST[6,7]는 S-W 알고리즘의 검색 속도 문제를 해결한 휴리스틱스 기반의 알고리즘이다. BLAST를 이용한 DNA 시퀀스 검색은 다음의 2 단계로 수행된다[20]. 첫 번째 단계에서는 질의 DNA 시퀀스의 모든 가능한 위치에 짧은 고정 길이 k의 슬라이딩 윈도우를 위치시켜 길이가 k인 모든 워드(word)들을 추출한다(그 결과, |Q|-k+1개의 워드가 얻어진다). 다음, 전체 데이타베이스를 스캔하여 워드들과 정확히 일치하는 k-터플(k-tuple)들을 찾아낸다. 이때 대응되는 워드와 k-터플을 시드(seed)라 한다. 두 번째 단계에서는 각 시드를 확장하여 미리 정해진 허용치 T 이상의 유사성을 가지는 유사 세그먼트 쌍을 찾아낸다. 다음, 이 들 유사 세그먼트 쌍을 이용하여 질의 시퀀스와 유사한 시퀀스들을 결과로서 반환한다. BLAST는 유사 허용치 T와 워드의 길이 k 값을 통하여 검색 속도와 검색 결과의 정확도를 조절할 수 있다. T와 k 값이 큰 경우, 유사하지 않은 많은 부분들을 사전에 필터링함으로써 검색 속도를 높일 수 있으나, 유사한 정도가 크지 않은 부분들을 최종 결과에 포함시킬 수 없다. 반면, T와 k 값이 작은 경우, 유사한 정도가 그리 크지 않은 부분들도 최종 결과에 포함시킬 수 있으나, 많은 비교 연산으로 인하여 검색 속도가 떨어진다. BLAST는 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.

한편 최근 BLAST의 검색 속도와 정확도를 개선할 수 있는 다양한 방식들[9,10]이 제안되었다. 그러나 이 들 방식 또한 데이타베이스의 순차 검색을 기본으로 하므로 데이타베이스 크기가 증가하는 경우 검색 효율이 저하하는 단점을 갖는다.

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[4,21,22], 다차원 인덱스 방식[23], 영속 트리 방식[16,17,24] 등으로 분류할 수 있다.

기존의 정보 검색 분야에 활용되는 역 인덱스(inverted index)를 기반으로 하는 인덱싱 기법으로 참고 문헌 [4,21,22]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 기존의 BLAST 등에 비하여 속도 향상 효과를 얻을 수 있다. 참고 문헌 [4]에서는 특히 압축 인덱스 구조를 사용하여 인덱스 파일이 과다하게 커지는 단점을 보완하고 있으나, 검색 결과의 정확도가 떨어지는 점 등이 단점으로 지적되고 있다.

참고 문헌 [23]에서는 웨이블렛(wavelet) 변환에 의하여 시퀀스 데이타베이스 내의 각 서브시퀀스를 다차원의 정수 공간으로 매핑한 후, 이 들을 다차원 인덱스 구조로 표현하고, 영역 질의 방식에 의하여 유사 시퀀스를 검색할 수 있는 MRS-인덱스 방식을 제안하고 있다. 제안된 방식은 작은 사이즈의 인덱스 구조를 이용하여 질의 처리 시 데이타베이스의 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 지적될 수 있다[5].

영속 트리 기반의 대표적인 인덱스 방식으로서 접미어 트리(suffix tree)[11]를 들 수 있다. 참고 문헌 [17]에서는 대용량의 DNA 시퀀스 데이타베이스를 위한 디스크 기반의 접미어 트리 구성 알고리즘을 제안하고 있으며, 참고 문헌 [24]에서는 인메모리 방식의 압축 접미어 배열[25]를 이용한 유사 시퀀스 검색 방식을 제안하고 있다. 또한 참고 문헌 [16]에서는 접미어 트리를 기본 인덱스 구조로 사용하여 S-W 알고리즘[8]과 같은 정확한 유사 검색결과를 얻는 새로운 방식을 제안하고 있다. 두 시퀀스 사이의 유사도 계산을 위하여 다이나믹 프로그래밍 A*-탐색 기법[26]을 적용하고 있으며, 접미어 트리 구조를 이용한 성능 개선 효과를 얻고 있다. 그러나 접미어 트리는 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 갖는다.

2.2 접미어 트리

접미어 트리(suffix tree)는 접미어 트라이(suffix trie)의 축약된 형태이다. 접미어 트라이는 주어진 시퀀스에 속하는 모든 접미어들을 저장하는 인덱스 구조로서, 에지는 시퀀스에 속하는 하나의 심볼을 라벨로 가지며, 루트로부터 단말 노드까지의 경로는 인덱싱의 대상이 되는 하나의 접미어에 해당된다[11]. 그림 1은 S1='ACGT'와 S2='ACT'의 두 개의 DNA 시퀀스에 대하여 접미어 트라이를 구성한 예를 나타낸 것이다. 여기에서 '$' 기호는 시퀀스의 끝을 나타내어, 각 접미어를

유일하게 구별하는데 사용된다. 예를 들어 그림 1에서 시퀀스 S1의 접미어 'CGT'는 루트로부터 (S1,1)의 라벨을 갖는 단말 노드까지의 경로로 표현된다. 단말 노드의 라벨은 시퀀스 번호와 오프셋을 나타낸다.

접미어 트리는 접미어 트라이에서 단 하나의 자식 노드만을 갖는 노드들의 경로를 하나의 에지로 표현한 형태이다. 접미어 트라이의 경우와 같이 루트로부터 단말 노드까지의 경로는 인덱싱의 대상이 되는 하나의 접미어에 해당하며, 루트로부터 단말 노드까지 경로상의 에지 라벨들을 모두 접합(concatenation)하면 접미어가 된다. 또한 루트노드로부터 한 내부 노드(internal node)까지 경로상의 에지 라벨들을 모두 접합한 결과는 그 내부 노드 아래에 존재하는 모든 단말 노드와 대응되는 접미어들의 최장 공통 접두어(longest common prefix)가 된다. 그림 2는 그림 1의 접미어 트라이를 접미어 트리로 바꾼 것이다. 접미어 트라이에 비하여 상당수의 중간 노드들이 제거되었음을 볼 수 있다.

접미어 트리는 다수의 시퀀스들을 인덱싱 하기 위하여 사용되며, 주어진 질의 시퀀스와 정확하게 매치되는 서브시퀀스의 위치를 신속하게 찾도록 하는데 유용하다. DNA 시퀀스 데이타베이스 S로부터 생성된 접미어 트리를 이용하여 질의 시퀀스 $Q(=q_1q_2...q_{|Q|})$의 출현 유무 및 출현 위치를 검색하는 방식은 다음과 같다. 접미어 트리의 루트 노드로부터 시작하여 질의 시퀀스와 일치하는 트리 내의 경로를 검색한다. 접미어 트리의 운행 결과, 질의 시퀀스와 일치하는 경로를 찾아 임의의 노드 N에 이르면 질의 시퀀스가 검색되었음을 의미하며, N이하의 모든 서브 트리에 존재하는 단말 노드 정보(시퀀스 번호, 오프셋)가 결과로 반환된다. 그러나 질의 시퀀스와 일치하는 경로를 찾지 못하면, DNA 시퀀스 데이타베이스 S에는 질의 시퀀스 Q가 서브시퀀스로 존재하지 않음을 의미하며 이를 결과로 알린다. 예를 들어 그림 2의 접미어 트리를 사용하여 질의 시퀀스 'AC'를 검색하는 과정은 다음과 같다. 루트 노드로부터 출발하여 질의 시퀀스 'AC'와 일치하는 경로를 따라 순행하면 루트의 가장 왼쪽 자식 노드에 이르게 된다. 따라서 질의 시퀀스의 검색에 성공하고 그 결과로서 해당 노드의 모든 단말 노드 정보 (S1, 0)와 (S2, 0)를 결과로 반환한다.

접미어 트리의 이론적 질의 검색 비용은 O(|Q|+p)으로 얻어진다[11]. 여기에서 |Q|는 질의 시퀀스의 길이를 나타내고, p는 검색된 결과의 개수를 나타낸다. 그러나 접미어 트리의 크기는 인덱싱 대상이 되는 시퀀스 데이타베이스의 길이에 선형적으로 비례하여 일반적으로 그 크기가 매우 크다. 실제적인 인덱스 크기는 각 노드 및 에지의 구현 방식 및 시퀀스 특성에 따라 달라진다. 최적의 구현 방식에 관한 많은 연구 결과가 보고되어 있으며[25,27-29], 대표적인 초기 논문으로 참고 문헌 [27]를 들 수 있다. 최근 참고 문헌 [29]에서는 인덱싱 대상의 시퀀스 문자당 13바이트까지로 구현 가능한 접미어 트리 최적화 방식을 제안하고 있으나, 이 들 알고리즘들도 디스크 기반으로 구현할 경우, 시퀀스 문자당 65바이트 정도가 필요하다[17].

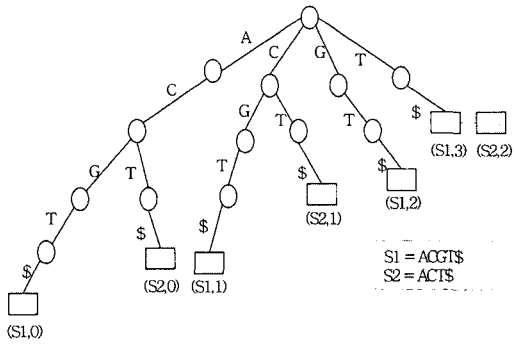

그림 1 접미어 트라이 구조

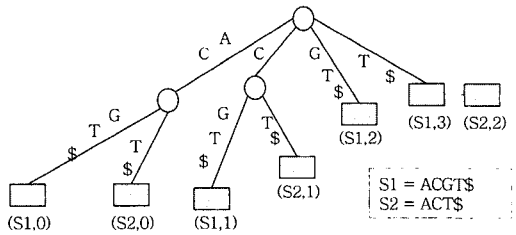

그림 2 접미어 트리 구조

3. 인덱싱 방안

본 장에서는 대규모 DNA 시퀀스 데이타베이스를 대상으로 하여 효율적인 DNA 시퀀스 검색을 지원하기 위한 인덱싱 방안에 대하여 논의한다. 제 3.1절에서는 기본 인덱스 구조로서 사용되는 트라이(trie)에 대해서 설명하고, 제 3.2절에서는 DNA 시퀀스로부터 이진 접미어 트라이를 구축하는 방안에 관하여 논의한다. 제 3.3절에서는 이진 접미어 트라이의 단말 노드를 저장하는 방안에 관하여 논의하고, 제 3.4절에서는 인덱스를 구축하는 알고리즘을 제시한다.

3.1 트라이

Retrieval을 어원으로 가지고 있는 트라이(trie)[11, 19]는 [30]에서 처음으로 개발되었으며 [31]에 의해 집중적으로 논의되었다. 트라이는 매우 단순한 구조를 가지고 있지만 검색 성능이 우수하기 때문에 문자열을 대상으로 하는 많은 응용에서 사용되고 있다. 트라이는 노

드가 정보를 가지지 않고 에지에 데이타가 저장되는 트리 구조로서, 각 에지는 하나의 문자를 가지며 루트로부터 단말까지의 경로는 인덱싱의 대상이 되는 하나의 키에 해당된다. 검색 연산시, 레벨 i에서의 서브 트라이 선택은 검색 키의 i번째 문자에 의해서 결정된다. 트라이 구조는 다음과 같은 특징을 가지고 있다.

① 모든 키의 공통 접두어는 오진 한 번만 저장된다. 따라서 데이타를 압축적으로 표현하게 된다.
② 트라이의 검색 시간은 트라이의 크기가 아니라 검색 키의 길이에 비례한다.
③ 트라이의 모양은 데이타의 입력 순서와 무관하며 주어지는 데이타 집합에 의해 유일하게 결정된다.
④ 트라이는 재구성(reorganization) 알고리즘이 필요하지 않다.

3.2 이진 접미어 트라이 구축 방안

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)[32]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. 그림 3은 이진 시퀀스 데이타 S1='001010'과 S2='110100'를 이진 트라이 구조로 표현한 예를 보이며, 그림 4는 이진 트라이 구조를 포인터를 사용하지 않는 이진 데이타로 표현한 방식의 예를 보인다.

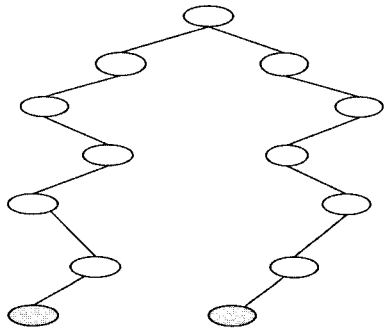

그림 3 이진 트라이 구조의 예

```
11
10 01
01 10
10 01
01 10
10 10
00 00
```

그림 4 포인터를 사용하는 않는 이진 트라이 표현의 예

본 연구에서는 포인터를 사용하지 않는 이진 트라이의 기본 개념을 활용하여 DNA 시퀀스 검색을 지원하는 인덱싱 구조를 제안한다. 이를 위해 DNA 시퀀스로부터 가능한 모든 접미어를 추출하여 이를 인덱싱의 대상으로 한다. 따라서 인덱스의 기본 구조로서 접미어 트라이[11] 구조를 사용한다.

접미어 트라이는 인덱싱 대상이 되는 접미어들이 많은 공통 접두어 서브시퀀스를 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 접두어 서브시퀀스를 가질 확률이 높아지므로, DNA 시퀀스를 대상으로 접미어 트라이 인덱스를 생성하면 기본적으로 높은 압축 효과를 기대할 수 있다.

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 예를 들어 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 접미어 트라이를 구성하면 공통 접두어 서브시퀀스가 생성될 확률이 더욱 높아지며, 따라서 생성된 트라이 인덱스 구조로부터 더욱 높은 압축율을 기대할 수 있다.

DNA 시퀀스로부터 얻어진 이진 접미어 시퀀스를 위에서 정의한 이진 트라이 구조에 삽입하여 디스크 기반의 인덱스를 생성하는 과정을 예를 이용하여 단계별로 설명하면 다음과 같다.

우선, 각 문자를 이진 표현한 예를 그림 5에 보인다. 여기에서 사용된 '$' 문자는 각 시퀀스를 유일하게 구별하기 위하여 사용된 특수 문자를 나타낸다. 예를 들어 S1='ACGT'와 S2='ACT'의 두 개의 DNA 시퀀스에 대한 인덱스를 구성하는 경우, 각 시퀀스로부터 모든 접미어 시퀀스를 추출하여 그림 5의 방식에 의하여 각 문자당 3비트를 할당하여 이를 이진 시퀀스로 변환하면 그림 6과 같은 결과를 얻을 수 있다. 여기에서 '$' 문자가 각 접미어 시퀀스를 유일하게 구분하기 위하여 사용됨을 알 수 있다.

다음, 인덱스 구성 효율을 증가시키기 위하여 각 이진 접미어 시퀀스를 정렬하여 순서대로 트라이 구조에 삽입한다. 트라이 구조에 새로운 임의의 길이의 이진 시퀀스를 삽입하면 새로이 삽입되는 시퀀스의 각 비트 정보는 기존 트라이 노드 구조의 기존 경로를 따라 가거나, 새로운 에지를 생성하여 새로운 노드를 형성한다. 이와 같은 노드 구조의 변화 과정은 2비트 노드 정보로 변환되어 이진 데이타로 저장된다. 예를 들어, 그림 6의 예에서 보인 각 이진 시퀀스를 정렬하여 이진 트라이 구조에 삽입하면 그림 7와 같은 이진 트라이 구조를 얻을 수 있으며, 이와 같은 이진 트라이 구조는 그림 8과 같이 이진 데이타로 표현된다.

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

그림 5 각 출현 문자의 이진 표현 예

추출된 접미어 시퀀스		이진 변환된 접미어 시퀀스
S1:	ACGT$	001010011101000
	CGT$	010011101000
	GT$	011101000
	T$	101000
S2:	ACT$	001010101000
	CT$	010101000
	T$	101000

그림 6 DNA 시퀀스로부터 추출된 접미어 시퀀스의 예

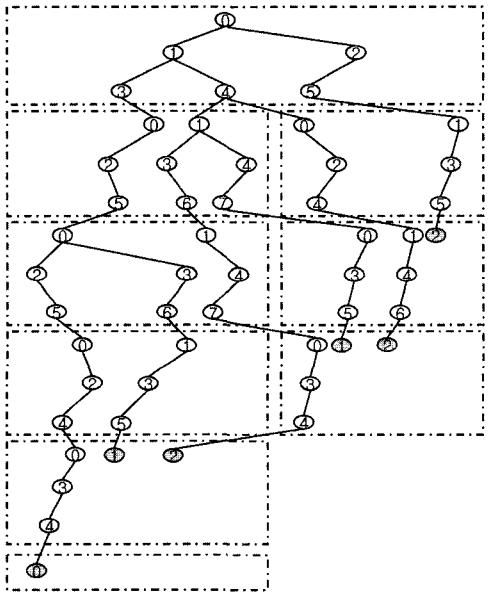

그림 7 접미어 트라이의구성 예

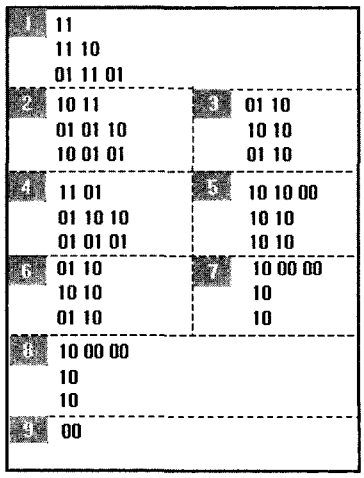

그림 8 접미어 트라이의 내부표현 예

이와 같이 생성되는 이진 트라이(즉, 이진 데이타)는 초기에는 주기억 장치에 생성되어 한 페이지 크기를 초과하면 바로 디스크에 쓰이게 된다. 단, 트라이 구조를 디스크에 분할 저장할 때 각 디스크 페이지에 저장할 최대 노드 레벨 수와 최대 노드 수를 미리 정하여 오버플로가 발생하지 않도록 한다. 또한 여기에서 주의할 점은 트라이 구조를 페이지에 저장하는 경우, 해당 페이지에 할당되어 들어온 에지에 의하여 형성되는 서브 트라이가 다른 페이지에 걸쳐 나누이지 않도록 보장하는 점이다. 이 가정은 검색 과정의 효율을 높이기 위하여 필요하다.

그림 7과 그림 8에서 보이는 점선은 디스크 내의 페이지 분할 저장 상황을 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. 단, 서브 트라이 구조가 서로 다른 페이지에 나뉘어 저장되는 것을 방지하기 위하여 페이지가 최대 노드 수에 못 미치는 노드 정보만을 가지고 있을 수 있다.

이와 같이 접미어 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 할당된 노드 정보를 나타내는 각 페이지에 유입된 에지의 수, 각 페이지로부터 나간 에지의 수 등으로부터 계산될 수 있다. 예를 들어 그림 8의 인덱스 구조의 페이지 교체를 위해서는 표 1과 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. 페이지 정보는 각 페이지의 노드 정보를 디스크에 기입한 후 생성된다.

표 1 페이지 테이블

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

3.3 단말 노드 저장 방안

접미어는 시퀀스 정보(시퀀스 식별자와 시퀀스 내의 오프셋)에 의해서 유일하게 구별된다. 접미어를 접미어 트라이에 저장할 때 접미어의 시퀀스 정보는 일반적으로 단말 노드에 저장된다. 그러나 제안되는 접미어 인덱스 상에서 각 노드는 2비트로 표현되므로 접미어의 시퀀스 정보는 접미어 인덱스와는 별도로 저장되어야 한다. 예를 들어 그림 8의 인덱스 구조에서 단말 정보 검색을 위하여는 각 페이지의 단말 노드 위치와 시퀀스 정보를 연결하는 다음 표 2와 같은 단말 정보를 별도로 저장하여야 한다. 여기에서 #Page는 페이지 번호를 나타내며, R/C는 단말 노드가 페이지 내에 출현한 위치(행/열) 정보를 나타내고, #Seq와 Offset은 시퀀스 식별자와 시퀀스 내의 출현 위치를 나타내는 오프셋 정보를 나타낸다.

표 2 단말 정보 테이블

#Page	R/C	#Seq	Offset
9	0 0	S1	0
8	0 1	S2	0
8	0 2	S1	1
7	0 1	S2	1
7	0 2	S1	2
5	0 2	S1	3
5	0 2	S2	2

제 4장에서 기술하는 접미어 트라이 기반의 질의 처리 방식에서는 질의 시퀀스와 일치하는 임의의 경로를 검색한 후, 해당 결과의 시퀀스 정보를 얻기 위하여 현재 노드 이하의 서브 트라이를 검색하여 모든 단말 노드와 단말 정보를 검색하는 과정을 필요로 한다. 그러나 이와 같은 단말 정보의 검색 과정은 인덱스 크기가 방대하고, 특히 질의 시퀀스의 길이가 짧은 경우, 검색 대상의 인덱스 공간이 매우 커질 수 있다는 단점이 있다.

본 연구에서는 단말 정보를 보다 효율적으로 검색하기 위하여 다차원 인덱스를 이용한 새로운 단말 정보 저장 방식을 사용한다. 다차원 인덱스 저장 방식에서는 접미어 시퀀스를 이진 비트 패턴으로 변환한 후, 이 값을 다차원 키 값으로 사용하여 단말 정보를 저장한다. 단, 이진 비트 패턴으로 변환된 접미어 시퀀스는 그 길이가 각각 다르므로, 다음과 같은 방식을 사용하여 각 시퀀스를 정해진 수 k개의 정수로 표현하여 k 차원의

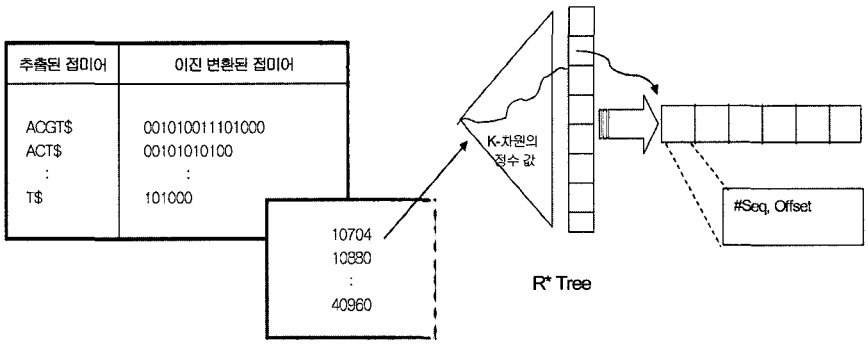

그림 9 다차원 인덱스를 이용한 단말 정보 저장의 예

다차원 인덱스 키 값으로 사용한다.

① 시퀀스의 이진 비트 패턴이 k개의 정수 표현보다 짧을 때: 해당 비트 패턴 뒤에 0을 k개의 정수 표현까지 계속 붙여 표현한 후, 이를 k 차원의 인덱스에 저장한다.

② 시퀀스의 이진비트 패턴이 k개의 정수 표현보다 길 때: 해당 비트 표현이 k개의 정수 표현까지로 표현될 수 있도록 k개의 정수 표현 이후의 뒷부분을 잘라 버린 후, 이를 k 차원의 인덱스에 저장한다.

S1='ACGT'와 S2='ACT'의 두 시퀀스의 단말 정보 저장을 위한 다차원 인덱스의 구성 예를 그림 9에 보인다. 각 접미어 시퀀스를 1차원의 2바이트 정수로 표현한 예로서, 예를 들어 접미어 시퀀스 'ACT$'는 이진 변환된 이진 문자열의 길이가 2바이트 정수 표현(16비트)보다 길이가 짧으므로 뒷부분에 '0'이 삽입되어, 1개의 정수 '10880'으로 표현되어 다차원 인덱스에 삽입된다.

3.4 인덱스 구성 절차

DNA 시퀀스를 위한 접미어 트라이 인덱스 구성 절차는 다음과 같은 단계로 이루어진다.

단계 1: 접미어 시퀀스 추출

유전체 데이타베이스 내의 각 DNA 시퀀스의 모든 접미어 시퀀스를 추출한다.

단계 2: 이진 접미어 시퀀스로 변환

추출된 모든 접미어 시퀀스에 대해 시퀀스를 구성하는 각 문자당 최소 비트를 할당하여 시퀀스를 이진 비트열로 변환한다. 이를 이진 접미어 시퀀스라 정의하고, 이를 인덱싱 대상으로 한다. 다음, 변환된 이진 접미어 시퀀스를 오름차순으로 정렬한다.

단계 3: 트라이 구성

정렬된 이진 접미어 시퀀스를 다음 과정에 의하여 순서대로 트라이에 삽입하여 이진 접미어 트라이를 구성한다.

① 시퀀스가 삽입되는 과정에 의하여 새로이 생성 또는 변경되는 노드 구조를 2비트 노드 정보로 해당 페이지 영역에 기록한다. 이때 한 페이지에 저장할 최대 트리 레벨 수와 최대 노드 수에 의하여 페이지 크기가 결정된다.

② 새로이 삽입되는 노드에 의하여 해당 페이지 영역 내에 오버플로우가 발생할 가능성이 있으면, 이 노드를 제외한 나머지 해당 페이지 영역을 디스크에 기록하고, 기록되지 않은 노드 정보로 해당 페이지 영역을 재구성한다. 디스크에 해당 페이지 영역을 기록할 때, 페이지 정보를 함께 기록한다.

③ 이진 접미어 시퀀스가 트라이 구조에 삽입된 후, 그 단말 정보를 정해진 k 개의 정수로 표현하여 k 차원의 인덱스에 저장한다.

4. 질의 처리 방안

본 장에서는 제 3장에서 제안한 인덱싱 기법을 이용한 질의 처리 방안에 대해 논의한다. 제 4.1절에서는 이진 접미어 트라이를 이용한 질의 처리 알고리즘을 보이고, 제 4.2절에서는 다차원 인덱스를 이용한 효율적인 단말 정보 검색 방안을 제시한다.

4.1 질의 처리 알고리즘

제안된 접미어 트라이 인덱스는 2비트의 노드 정보만을 저장할 뿐 노드와 노드를 연결하는 포인터 정보와 노드 레벨 정보는 저장하지 않는다. 그러므로 주어진 질의 시퀀스와 일치하는 서브 시퀀스를 인덱스 내에서 검색하기 위해서는 해당 페이지를 검색한 후, 페이지 내의 노드 레벨 정보 및 노드의 연결 정보 등을 연산과정을 통하여 얻어야 한다.

아래에 명시된 알고리즘 Search-Trie는 접미어 트라이 인덱스 T를 이용하여 질의 Q를 만족하는 서브시퀀스들을 검색하는 알고리즘을 나타낸다. 여기에서 질의 Q는 질의 시퀀스를 이진 비트열로 변환한 것임에 유의하여야 한다. 또한 트라이 인덱스의 페이지 분할에 관련된 정보는 페이지 테이블 P에 저장되어 있다고 가정한다.

알고리즘에서는 페이지 내의 트라이 구조를 파악하기 위하여 다음과 같은 4개의 변수를 이용한다. 우선, size 변수는 각 노드 레벨 내의 노드 수를 파악하기 위하여 사용되며, 현재 노드 정보가 '11'이면 다음 레벨에서는 노드 수가 하나 증가하는 의미를 가지므로, size 변수의 값을 1만큼 증가시키고, 노드 정보가 '00'이면 현재 노드로 경로가 종단됨을 의미하므로 size 변수의 값을 1만큼 감소시킨다. last 변수는 각 노드 레벨에서의 마지막 노드의 위치를 나타내며, 다음 노드 레벨의 last 변수의 값은 현재 레벨의 last 변수의 값에 size 변수의 값을 더하여 얻을 수 있다. next 변수는 다음 노드 레벨에서 입력 질의 비트와 실제로 비교를 수행하여야 하는 노드를 지정하는 변수이고, 이 값을 얻기 위하여 다음의 srch 변수를 중간 변수로 사용한다. srch 변수는 현재 노드 레벨의 처음 노드로부터 질의 비트와 매치되는 노드의 해당 비트까지 출현한 모든 이진 비트 값 '1'의 수를 더한 값을 변수 값으로 가지며, 다음 노드 레벨의 노드 수를 카운팅하는 역할을 담당한다. 따라서 다음 레벨의 next 변수 값은 현재 레벨의 last 변수 값에 srch 변수 값을 더하여 얻을 수 있다.

Search-Trie의 동작과정을 간략히 설명하면 다음과 같다. 여기에서 전체 트라이 인덱스는 p_Height 개의 페이지 레벨로 나뉘어 있으며, 각 페이지 레벨은 다시 n_Height 개의 노드 레벨로 이루어져 있다고 가정한다. 우선, 트라이 인덱스의 처음 페이지의 처음 노드는 루트 노드를 나타내므로 이에 맞추어 각 변수를 초기화 한다(Line 1). 전체 알고리즘은 각 페이지 레벨과 노드 레벨에 대한 이중의 For 문으로 이루어져 있다. 외부 For 문(Line 2-14) 내의 Line 3-5에서는 다음 레벨로의 페이지 교체가 이루어지는 과정을 나타낸 것으로, 함수 Page_Change()에서는 페이지 정보 P를 참조하여(Top, Bottom 값을 이용하여) 다음 검색 대상의 페이지를 찾아서 이를 읽어 들인다. 다음, 트라이 검색을 진행하기 위한 각 변수의 값이 재설정 된다. 내부 For 문(Line 6-14)은 노드 레벨에 대한 처리 과정을 나타내는 것으로 다음 4단계 과정으로 이루어져 있다. 처음 1단계로서 Line 7-9에서는 next 노드 직전까지 노드 정보를 순차적으로 읽어 들이며 size 값과 srch 값을 변경시키고 있다. 다음 2단계인 Line 10-12에서는 next 노드와 질의 비트를 비교, 처리하는 과정으로 Line 10은 매치에 실패한 경우를 나타내고, Line 11은 현재 노드 매치에 성공했고, 해당 질의 비트가 질의의 마지막 비트인 경우를 나타낸다. 질의의 마지막 비트까지 매치에 성공하면 find_answers() 함수가 호출되어, 해당 노드의 서브 트라이에 존재하는 모든 단말 노드로부터 시퀀스 정보(시퀀스 번호, 오프셋)를 얻게 된다. Line 12에서는 다음 입력 비트를 읽어 들이고 다시 size 값과 srch 값을 변경시키는 과정을 나타낸다. 다음, 3단계인 Line 13에서는 last 노드 직전까지 노드를 순차적으로 읽어 들이며 size 값을 변경시키고 있다. 마지막 4단계인 Line 14에서는 현재 노드 레벨이 현재 페이지의 마지막 노드 레벨이 아니라면 다음 노드 레벨 수행을 위하여 next, srch, last 변수 값을 재설정하고 있다.

Algorithm Search_Trie

Input : Trie index T, Query Q, Page Info P
Output : set of answers

```
1. srch := 0, next := 0, size := 1, last := 0;
2. For (pageLevel := 0; pageLevel < p_Height;
   pageLevel++) do
3.    if (pageLevel > 0) then {
4.                    Page_Change(),
5.                    reset size, last, next, srch } ;
6.    For (nodeLevel := 0; nodeLevel < n_Height;
      nodeLevel++) do
7.       while (isBefore(next)) do {
8.                                  increment srch,
9.                                  update size } ;
10.      if (!match(node(next), query_bit)) return('No
         Match') ;
11.      if (isLast(query_bit)) return(find_answers()) ;
12.      get(query_bit), increment srch, update size ;
13.      while (isBefore(last)) do { update size } ;
14.      if (nodeLevel < n_Height - 1) then { reset
         next, srch, last } ;
```

알고리즘의 간단한 수행 방식을 보이기 위하여 제 3장에서 예제로 사용한 접미어 트라이 인덱스(그림 8 참조)를 이용한 질의 시퀀스 'T'의 검색 과정을 일부 도식화하여 나타내면 다음과 같다(그림 10). 단, 여기에서는 단말 정보 검색을 위하여 표 2에 보인 간단한 단말 정보 테이블만을 사용하는 경우를 보인다.

질의 시퀀스 'T'는 이진 비트열 '101'로 변환되며, 1번

[1번 페이지]

position	bitPair	query	srch	next	size	last
			0	0	1	0
0	11	1	2,0	2	2	2
1	11		2		3	
2	10	0	3,0	5	3	5
3	01		1		3	
4	11		3		4	
5	01	1	4,0	9	4	9

[3번 페이지로 교체]

			0	1	2	1
0	01					
1	10					
2	10					
3	10					
4	01					
5	10					

그림 10 질의 처리 과정의 예

페이지의 처음 노드(루트 노드)로부터 검색을 수행하여, 2번 노드와 5번 노드와의 비교를 수행한 후, 페이지를 3번으로 교체하고, 각 변수들의 값을 재설정한다. 3번 페이지로 교체되면서 이미, 질의 시퀀스 검색에 성공하였으므로 3번 페이지의 1번 노드 이후의 모든 서브 트라이를 검색하여 서브 시퀀스의 출현을 나타내는 단말 노드를 검색한다. 서브 트라이의 단말 노드 검색 과정은 깊이 우선 방향으로 진행되며, 노드 정보가 '00'인 노드에 도달하면 단말 정보 테이블을 검색하여 해당 단말 정보에 수록된 시퀀스 정보를 얻게 된다. 이 예에서는 5번 페이지의 2번 노드에서 단말 노드가 검색되어 단말 정보 테이블로부터 시퀀스 번호 S1(오프셋 3)과 S2(오프셋 2)를 최종 검색결과로 얻게 된다.

4.2 다차원 인덱스를 이용한 단말 노드 정보 검색

제 4.1절에서 보인 Search_Trie 알고리즘에서는 질의 처리 과정 중, 접미어 트라이의 중간 노드 N에서 질의 시퀀스가 검색 완료되면, 해당 노드 N의 서브 트라이에 존재하는 모든 단말 정보를 가져오는 과정이 필요하다. 바로 find_answers() 함수가 단말 정보를 검색하는 역할을 담당한다.

제 3.3절에서 제안한 다차원 인덱스는 서브 트라이의 중간 노드 검색 과정 등을 필요로 하지 않고 직접적인 단말 정보의 검색을 가능하게 한다. 다차원 인덱스는 접미어 트라이를 이용하여 질의 시퀀스의 검색을 완료한 후, 직접 단말 정보를 검색해 내는 방식으로서, 단말 정보의 검색 방식은 다음과 같다. 트라이 인덱스의 루트로부터 중간 노드 N을 연결하는 경로 p가 주어진 질의를 만족한다면, 그 경로의 길이에 따라 다음 3가지 중 하나를 선택하여 질의를 처리함으로써, 단말 정보를 얻을 수 있다.

① 경로 p의 길이가 k개의 정수 표현 보다 짧을 때: 경로 p 뒤에 0을 계속 붙여 만든 k-정수 값을 p0라고 하고, p뒤에 1을 계속 붙여 만든 k-정수 값을 p1이라고 하자. 다차원 인덱스를 검색하여 p0와 p1 사이에 존재하는 모든 단말 노드들을 검색한다.

② 경로 p의 길이가 정확하게 k개의 정수 표현 일 경우: 다차원 인덱스를 검색하여 경로 p를 k-정수로 표현한 값을 가지는 모든 단말 노드들을 검색한다.

③ 경로 p의 길이가 k개의 정수 표현 보다 길 때: 경로 p를 k-정수로 표현한 값을 pt 라고 하자. 다차원 인덱스를 검색하여 pt 값을 가지는 모든 단말 노드들을 검색한다. 그 후, 후처리 과정에 의하여 잘못 검색된 단말 노드를 제거한다.

앞에서 예로 사용한 질의 시퀀스 'T'에 대하여 다차원 인덱스를 이용한 단말 정보 검색 과정을 그림 11에 보인다. 질의 시퀀스 'T'를 이진 변환하면 '101'이고, 다차원 인덱스로는 2바이트 정수로 이루어진 1차원 인덱스를 이용한다고 가정한다(그림 9 참조). 우선, 질의 시퀀스와 일치하는 노드를 이진 접미어 트라이 인덱스에서 검색한 다음, 다차원 인덱스에서 단말 정보를 검색한다. 이진 접미어 트라이 인덱스의 루트 노드부터 질의 시퀀스와 일치하는 중간 노드 N까지의 경로로부터 얻어진 p='101'이 2바이트로 표현되는 정수 비트 패턴보다 길이가 짧으므로 경로 p 뒤에 0을 붙여 생성한 p0='1010000000000000'와 1을 붙여 생성한 p1='1011111111111111'를 얻는다. 다음, 다차원 인덱스에 대하여 p0와 p1 사이의 영역 질의를 수행하여 시퀀스 번호 S1(오프셋 3)과 S2(오프셋 2)를 검색 결과로 얻게 된다.

5. 성능 평가

본 장에서는 실험에 의한 성능 분석을 통하여 제안하는 기법의 우수성을 규명한다. 제 5.1절에서는 실험 환경을 설명하고, 제 5.2절에서는 실험 결과를 분석한다.

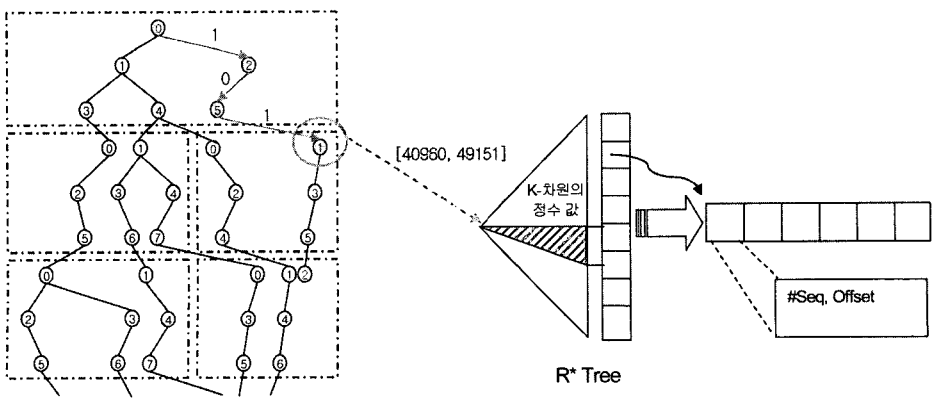

그림 11 다차원 인덱스를 이용한 단말 노드 정보 검색의 예

5.1 실험 환경

본 실험에서는 GenBank[33]로부터 다운 받은 human chromosome 18, 19, 21의 시퀀스를 이용한다. 실험에는 세 개의 서로 다른 크기의 human chromosome 18 시퀀스(1.07Mbp, 2.16Mbp, 4.22Mbp)와, 43.3Mbp의 human chromosome 21 시퀀스, 72.3Mbp의 human chromosome 19 시퀀스의 다섯 가지 데이타 세트를 사용한다. 질의 시퀀스는 각 실험 데이타로부터 랜덤 방식에 의하여 임의의 길이의 질의 시퀀스를 추출하여 사용한다.

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 서로 다른 접미어 시퀀스의 구별을 위하여 사용되는 '$' 문자를 포함하여 8가지 문자를 처리 대상으로 한다.

성능 평가는 다음 네 가지의 서로 다른 기법을 대상으로 한다. Trie-Rtree는 본 논문에서 제안한 트라이 인덱스 검색 방식으로서, 단말 노드 검색을 위하여 별도의 다차원 인덱스를 사용하는 방식이다. 다차원 인덱스로는 현재 시퀀스 데이타베이스 분야에서 널리 사용되고 있는 R*-트리[34]를 사용하여 실험한다. 사용된 R*-트리는 Maryland 대학의 Faloutsos 교수 팀에서 개발한 R*-tree Version 2.0을 사용하며, 본 실험에서는 단말 정보를 2차원 데이타로 표현하여 저장한다. 성능 비교를 위한 기존의 기법으로서 순차 검색 방식인 Seqscan, 접미어 트리를 이용한 검색 방식인 Suffix, 별도의 단말 노드 검색 기법을 사용하지 않는 트라이 인덱스 검색 방식인 Trie-Naive의 세가지 방식을 사용한다. Suffix에서는 참고 문헌 [35]에서 제안된 incremental disk-based 접미어 트리 생성 방식을 사용하며, 생성된 접미어 트리는 각 노드 당 최대 32바이트를 필요로 한다.

실험을 위한 하드웨어 플랫폼으로는 Windows 2000 Server를 운영체제로 사용하고, 1GB의 주기억 장치, 40GB 디스크를 갖는 Pentium IV 2GHz의 PC를 사용한다.

5.2 실험 결과 및 분석

제안된 기법의 성능을 평가하기 위하여 인덱스의 크기 및 질의 처리 시간을 기존 방식과 실험을 통하여 비교 분석한다.

먼저 실험 1에서는 본 논문에서 제안한 Trie-Rtree의 인덱스 크기에 대하여 Suffix와 Trie-Naive를 비교, 평가한다. 표 3에 비교 대상의 세 가지 기법에 대하여 인덱스의 크기를 구성 요소별로 분리, 측정한 결과를 보인다. 본 실험에서는 페이지 크기로 4K를 가정하였다. Suffix의 인덱스는 단일 인덱스 구조로서 내부 노드와 단말 노드로 구성된다. Trie-Naive의 인덱스는 이진 접미어 트라이 인덱스와 페이지 테이블, 단말 정보 테이블로 구성된다. 페이지 테이블은 트라이 인덱스의 페이지 수와 같은 데이타 항목을 가지므로 그 크기가 그리 크지 않지만, 단말 정보 테이블은 입력 시퀀스 데이타의 모든 접미어 시퀀스의 개수와 같은 데이타 항목을 가지므로 그 크기가 비교적 크다. 한편, Trie-Rtree의 인덱스는 Trie-Naive와 동일한 이진 접미어 트라이 인덱스와 페이지 테이블로 구성되나, 단말 정보 검색을 위하여 별도의 R-트리 인덱스를 필요로 한다.

그림 12에 이 들 방식에 대하여 데이타 크기의 변화에 따른 인덱스의 크기 변화를 비교한 결과를 보인다. 실험 결과로부터 Suffix, Trie-Naive, Trie-Rtree의 세 방식 모두 데이타의 크기 변화에 따라 거의 선형적으로 인덱스 크기가 비례하여 증가함을 알 수 있다. 그러나, 절대 크기를 비교 할 경우 Trie-Naive는 Suffix에 비하여 약 48%의 저장 공간 감소 효과를 나타내고 있으며, Trie-Rtree는 Suffix에 비하여 약 24%까지의 저장 공간 감소 효과를 나타내는 것으로 나타났다.

실험 2에서는 본 논문에서 제안한 기법의 질의 처리 시간을 비교 대상의 세 가지 방식과 비교, 평가한다. 표 4에 비교 대상의 네 가지 기법에 대하여 질의 처리 시간을 측정한 결과를 보인다. 실험 데이타로는 43.3Mbp의 human chromosome 21 시퀀스를 사용하였으며, 질의 길이 변화에 따르는 질의 처리 시간을 비교한 결과이다. 측정 시간은 주어진 질의에 대하여 질의 시퀀스가

표 3 인덱스 크기 비교

Data Size	Trie-Rtree			Trie-Naive			Suffix
	trie_idx	page_tbl	R_tree	trie_idx	page_tbl	leaf_tbl	
1.07Mbp	15.9M	80.3K	30.5M	15.9M	80.3K	12.7M	54.5M
2.16Mbp	31M	155K	59.6M	31M	155K	25.6M	108M
4.22Mbp	59.9M	300K	116.5M	59.9M	300K	49.7M	209M
43.3Mbp	577M	2.82M	1,132M	577M	2.82M	517M	2.07G
72.3Mbp	878M	4.29M	1,637M	878M	4.29M	858M	3.31G

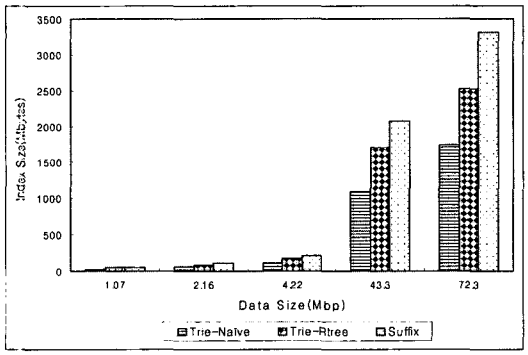

그림 12 데이타 크기 변화에 따른 인덱스 크기 비교

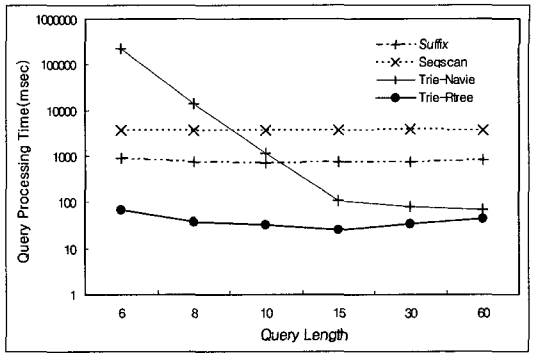

그림 13 질의 길이 변화에 따른 질의 처리 시간 비교

출현한 시퀀스의 번호와 오프셋 값을 반환하는 총시간을 의미하며, 여기에서 괄호 안의 값은 후처리 시간을 나타낸다. 후처리 시간이란 질의 처리 과정 중 접미어 트리(혹은 이진 접미어 트라이)의 중간 노드 N에서 질의 시퀀스가 검색 완료되어 해당 노드 N의 단말 노드 정보를 가져오는데 걸리는 시간을 의미한다. 실험 결과로부터 Suffix의 후처리 시간은 질의 길이가 증가함에 따라 감소하나, 어느 정도 길이 이상이 되면 전체 질의 처리 시간에 거의 영향을 미치지 않음을 알 수 있다. 한편, Trie-Naive와 Trie-Rtree의 경우, 질의 길이가 짧으면 후처리 시간이 전체 질의 처리 시간의 대부분을 차지하며, Trie-Naive에서는 특히 후처리 시간이 매우 많이 걸림을 알 수 있다. 그림 13은 질의 길이 변화에 따르는 질의 처리 시간(총시간)을 비교한 결과를 그래프로 표현한 것이다. Y축은 로그 스케일링 되어 있으며, 측정 단위는 msec이다.

실험 결과에 의하면 Trie-Naive는 질의 길이가 짧은 경우, 단말 노드 검색을 위한 트라이 인덱스 검색 시간이 증가하여 전체 검색 시간이 매우 많이 걸리지만 질의 길이가 긴 경우, 질의 처리 시간이 현저히 감소함을 볼 수 있다. 반면에 Trie-Rtree는 질의 길이에 무관하게 매우 좋은 검색 효율을 보이고 있으며, Suffix에 비하여 약 13배에서 29배까지의 성능 개선 효과를 나타내고, Seqscan에 비하여 약 54배에서 145배까지의 성능

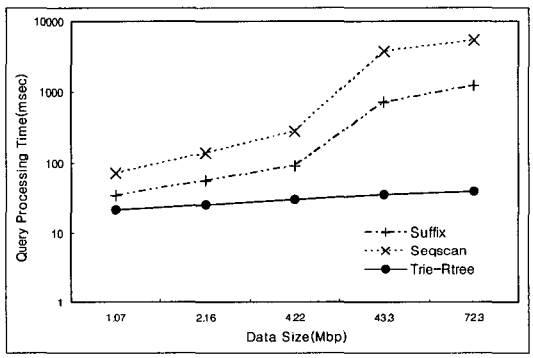

그림 14 데이타 크기 변화에 따른 질의 처리 시간 비교 (질의 길이 10의 경우)

개선 효과를 나타내는 것으로 나타났다.

실험 3에서는 데이타 크기의 변화에 따른 질의 처리 시간의 변화를 비교, 평가한다. 그림 14는 데이타 크기 증가에 따르는 질의 처리 시간(총시간)을 비교한 결과를 그래프로 표현한 것이다. Y축은 로그 스케일링 되어 있으며, 측정 단위는 msec이다. 여기에서는 질의 길이를 10으로 고정한 실험 결과를 보이며, Trie-Naive의 경우, 질의 길이가 짧은 경우 후처리를 위한 검색 시간이 많이 소요되므로 비교 대상에서 제외하였다. 실험 결과에 의하면 Seqscan은 데이타 크기에 비례하여 그 처리 시간이 선형적으로 증가하고 있음을 알 수 있다. Suffix

표 4 질의 처리 시간 비교

Query Length	Query Processing Time(msec)			
	Trie Rtree	Trie-Naive	Suffix	Seqscan
6	68.1(62.1)	230,423.5(230,417.5)	919.2(256.2)	3702.7
8	37.3(24.2)	13,612.4(13,599.3)	741.3(14.5)	3701.9
10	33(19.2)	1,152.4(1,138.6)	717,8(1.7)	3681.1
15	25.1(9.4)	104.9(89.2)	726.8(0.1)	3643.6
30	34.7(8.1)	79.4(52.8)	729.8(0)	3761.6
60	43.7(8.2)	67.5(32)	823.4(0)	3677.5

역시 데이타 크기가 증가함에 따라 질의 처리 시간이 증가하고 있으나, Trie-Rtree는 데이타 크기의 변화에 거의 무관하게 빠른 검색 시간을 보이고 있음을 볼 수 있다. 이와 같이 Trie-Rtree의 검색 성능이 Suffix에 비하여 향상된 이유는 다음과 같다. 우선, Suffix에 비하여 Trie-Rtree의 트라이 인덱스 크기가 감소하였다. 다음, Trie-Rtree에서 사용된 이진 트라이 인덱스와 R^*-트리는 관련 데이타들을 근접 디스크 내에 적절히 저장하는 최적화된 디스크 기반의 인덱스 구조로서 디스크 기반의 Suffix 보다 효율적인 디스크 액세스를 지원하는 인덱스 구조이다.

6. 결 론

DNA 시퀀스 검색은 분자 생물학 분야에서 사용되는 매우 중요한 연산이다. DNA 시퀀스 데이타베이스는 매우 큰 용량을 가지므로 DNA 시퀀스 검색의 효율적인 처리를 위해서는 좋은 인덱스의 사용이 필요하다. 본 논문에서는 DNA 시퀀스 검색을 위하여 기존에 제안된 접미어 트리가 가지는 저장 공간, 검색 성능, DBMS와의 통합 등의 문제점들을 지적하고, 이러한 문제점을 해결할 수 있는 새로운 인덱스를 제안하였다.

제안된 인덱스는 포인터 없이 트라이를 비트 스트링으로 표현하는 구조를 기본 구조로서 갖는다. 이러한 구조의 장점은 저장 공간의 오버헤드를 크게 줄일 수 있다는 점과 전체 구조를 페이지 기반으로 형성시킬 수 있다는 점이다. 또한, DNA 시퀀스 검색 시 검색 대상이 되는 트라이의 단말 노드를 신속하게 찾을 수 있도록 다차원 인덱스 구조를 보조 자료 구조로서 사용한다. 제안된 인덱스 구조를 사용함으로써 기존의 접미어 트리가 가지던 문제점들을 모두 해결할 수 있다. 또한, 제안된 인덱스를 이용하여 DNA 시퀀스 검색을 효과적으로 처리하는 질의 처리 알고리즘을 제시하였다.

제안된 기법의 우수성을 검증하기 위하여, 실험을 통한 성능 평가를 수행하였다. 실험 결과에 의하면, 제안된 인덱스는 기존의 접미어 트리와 비교하여 더 작은 저장 공간을 가지고도 13배에서 29배까지의 검색 성능의 개선 효과를 가지는 것으로 나타났다.

본 논문에서는 DNA 시퀀스 데이타베이스를 대상으로 하여 질의 시퀀스의 정확한 출현 위치를 검색하는 효율적인 인덱스 구조 및 질의 처리 알고리즘을 제안하였다. 제안된 방식은 다음 두 가지 방식으로 확장되어 유사 시퀀스 검색 문제에 적용될 수 있다. 현재 연구를 수행 중인 확장 방식과 기대효과는 다음과 같다.

① 제안된 Trie-Rtree 방식을 사용하여 질의의 정확한 출현 위치를 검색한 후, 데이타베이스 내의 질의의 출현 영역을 기반으로 유사 부분을 점진적으로 확장하여 유사도 T 이상의 서브 시퀀스를 검색한다. 여기에서 유사 부분을 점진적으로 확장하여 유사 서브 시퀀스를 검색하는 과정은 BLAST의 2단계 방법과 동일한 방식을 사용함을 가정하고 있으며, 따라서 전체적인 유사 검색 알고리즘의 고속화를 기대할 수 있다.

② 제안된 트라이 인덱스를 순행하며 다이나믹 프로그래밍 기법에 의하여 유사도 T 이상의 서브 시퀀스를 직접 구할 수 있도록 알고리즘을 확장한다. 단, 다이나믹 프로그래밍 기법의 직접적인 적용으로 인한 속도 저하 문제를 해결하기 위하여 효율적인 트리 순행 방식과 가지치기(pruning) 방식을 개발하여 시간과 정확도를 고려한 효율적인 검색 방식을 제공한다.

참 고 문 헌

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

[2] R. S. C. Goble, P. Baker, and Brass, "A Classification of tasks in bioinformatics," Bioinformatics, Vol. 17, No. 2, pp. 180-188, 2001.

[3] D. A. Benson, M. S. Boguski, D. J. Lipman, J. Ostell, and B. F. Quellette, "Genbank," Nucleic Acids Research, Vol. 26, No. 1, pp. 1-7, 1998.

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

[5] 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.

[6] 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.

[7] 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.

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

[9] J. Buhler, "Efficient Large-Scale Sequence Comparison by Local-Sensitive Hashing," Bioinformatics, Vol. 17, pp. 419-428, 2001.

[10] B. Ma, J. Tromp, and M. Li, "Patternhunter: Faster and more Sensitive Homology Search," Bioinformatics, Vol 18, pp. 440-445, 2002.

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

[12] 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.
[13] S. Kurtz, C. Schleiermacher, "REPuter: fast computation of maximal repeats in complete genomes," Bioinformatics, Vol. 15, No. 5, pp.426-427, 1999.
[14] G. Navarro and R. Baeza-Yates, "A new indexing method for approximate string matching," In Proceedings of Combinatorial Pattern Matching (CPM99), Lecture Notes in Computer Science, 1645, Springer, pp.163-185, 1999.
[15] E. Ukkonen, "Approximate string matching over suffix trees," In Proceedings of Combinatorial Pattern Matching (CPM93), Lecture Notes in Computer Science, 684, Springer, pp. 228-242, 1999.
[16] 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.
[17] 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.
[18] H. Wang et al., "BLAST++: A Tool for BLASTing Queries in Batches," In Proceedings First Asia-Pacific Bioinformatics Conference, pp. 71-79, 2003.
[19] E. Horowitz, S. Sahni, and S. Anderson-Freed, Fundamentals of Data Structures in C, Computer Science Press, 1993.
[20] D. W. Mount, Bioinformatics: Sequence and Genome Analysis, Cold Spring Harbor Laboratory Press, 2001.
[21] 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.
[22] 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.
[23] T. Kahveci and A. K. Singh, "An Efficient Index Structure for String Databases," In Proceedings of the 27th VLDB Conference, pp. 351-360, 2001.
[24] K. Sadakane and T. Shibuya, "Indexing huge genome sequences for solving various problems," In Proceedings of the 12th Genome Informatics, pp. 175-183, 2001.
[25] U. Manber and G. Myers, "Suffix Arrays: A New Method for On-Line String Searches," SIAM J. Comput., Vol. 22, No. 5, pp. 935-948, 1993.
[26] K. Kelly and P. Labute, "The A* Search and Applications to Sequence Alignment," http://www.chemcomp.com/article/astar.htm, 1996.
[27] E. M. McCreight, "A Space-Economic Suffix Tree Construction Algorithm," JACM, Vol. 23, No. 2, pp. 262-272, 1976.
[28] J. Kar kkainen and E. Ukkonen, "Sparse Suffix Trees," In Proceedings of COCOON, pp. 219-230, 1996.
[29] S. Kurtz, Reducing the Space Requirement of Suffix Trees. Softw. Pract. Exp., Vol 29, pp. 1149-1171, 1999.
[30] R. De La Briandais, "File searching using variable length keys," In Proceedings of Western Joint Computer Conference, Vol. 15, pp. 295-298, 1959.
[31] D. E. Knuth, Sorting and Searching, The Art of Computer Programming: Vol. 3, Addison-Wesley, 1973.
[32] 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.
[33] http://www.ncbi.nlm.nih.gov
[34] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles," In Proceedings of ACM SIGMOD International Conference on Management of Data, pp. 322-331, 1990.
[35] P. Bieganski, J. Riedl, J. V. Carlis, "Generalized suffix trees for biological sequence data: applications and implementation," In Proceedings of 27th Hawaii International Conference on System Sciences, Vol. 5, pp. 35-44, 1994.

원 정 임
정보과학회논문지 : 데이타베이스
제 31 권 제 5 호 참조

윤 지 희
정보과학회논문지 : 데이타베이스
제 31 권 제 5 호 참조

박 상 현
정보과학회논문지 : 데이타베이스
제 31 권 제 5 호 참조

김 상 욱
정보과학회논문지 : 데이타베이스
제 31 권 제 5 호 참조