가중치를 갖는 문자의 개수를 서명으로 이용한 DNA 인덱스 구조

김우철[O]T 민준기 박상현
연세대학교 컴퓨터과학과
{twelvepp[O], jkmin, sanghyun}@cs.yonsei.ac.kr

A DNA Index Structure Using Signature by Weighted Number of Characters

Woo-Cheol Kim[0], Joon-Gi Min, Sanghyun Park
Dept. of Computer Science, Yonsei University

요 약

우리는 대규모의 유전자 데이터베이스에서 원하는 패턴을 빠르고 정확하게 찾고 싶어한다. 하지만 지금까지 나온 대부분의 검색방법들은 인덱스의 크기를 실제 데이터보다 훨씬 크게 만들어 사용해왔다. 그런 방법들은 기하급수적으로 증가하고 있는 데이터를 처리하는 데는 비효율적이다. 따라서 인덱스 크기를 실제 데이터보다 작게 만들면서도 원하는 패턴을 빨리 찾을 수 있는 효율적인 방법이 필요하다. 이렇게 하기 위해서는 일정한 크기의 데이터를 작은 크기의 데이터로 줄인 후, 이 데이터를 이용하여 인덱스를 만들어야 한다. 이 논문에서는 일정한 크기의 문자열(=윈도우)을 작은 크기의 숫자들(=서명)로 표현해서 인덱스를 구축한 후, 이를 이용해 우리가 원하는 패턴을 최소한의 디스크 접근을 통해 빠르게 찾을 수 있는 방법을 제시한다.

1. 서 론

과학이 발전하면서 이용할 수 있는 데이터는 기하급수적으로 증가하고 있다. 특히 인터넷의 데이터나 유전자 데이터 같은 대규모의 문자열 데이터베이스에서 우리가 원하는 정보를 빠르고 정확하게 찾을 수 있는 방법은 점점 중요해지고 있다.

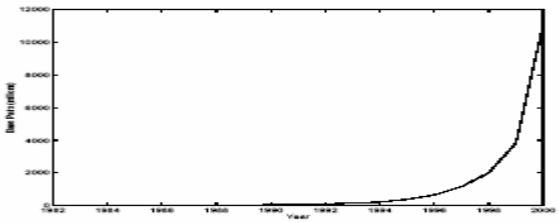

그림 1. NCBI데이터베이스 크기 증가곡선[4]

그러나 일반적인 문자열 검색 방법은 메모리 안에서의 검색을 가정하고 있다. 또한 인덱스를 이용한 대부분의 방법들은 실제 데이터보다 지나치게 큰 인덱스를 가지고 있다.

그런 방법들은 메모리 크기가 제한적이고 데이터가 기하급수적으로 증가하는 것을 고려할 때 효율적이라고 하기는 힘들 것이다. 그래서 이 논문에서는 대량의 문자열 데이터와 제한적인 메모리 크기 안에서 원하는 패턴을 효율적으로 찾아 낼 수 있는 작고 효율적인 인덱스 구조를 제시한다.

즉, 일정한 크기의 문자열 내에서 각 문자가 나타난 개수를 그 문자열의 서명으로 이용해 인덱스를 만드는 방법을 먼저 제시한다. 다음으로 인덱스의 데이터를 공간상에 골고루 분산시키기 위해서 문자가 나타난 위치에 가중치를 부여하는 방법을 제시한다.

이 논문에서는 기본적으로 3가지 문자열 검색 문제를 풀어보고자 한다. 첫 번째는 텍스트와 패턴이 정확히 일치하는 경우를 찾는 exact-match문제이며, 두 번째는 wildcard문자가 텍스트나 패턴에 나타나는 것을 허용하는 wildcard문제이며, 세 번째는 텍스트와 패턴을 비교할 때 최대 k개의 불일치를 허용하는 k-mismatch문제이다.

2. 관련 연구

문자열 검색 문제에 가장 많이 사용되는 자료구조는 서픽스 트리(Suffix Tree)[1]이다. 텍스트의 길이가 m이라고 할 때, 서픽스 트리의 크기는 이론 상으로 $O(m^2)$이고 실제로는 데이터베이스의 크기의 10배 이상 되기 때문에 데이터의 크기가 큰 경우에는 효율적이지 않은 구조이다. 예를 들어 참고 문헌[3]에 의하면, 286M 염기를 가지는 DNA 시퀀스를 대상으로 구성된 서픽스 트리의 크기는 19G 바이트로 나타났다.

작은 인덱스 구조를 이용한 효율적인 문자열 데이터베이스 검색방법은 [4]에 제시되어 있다. [4]에서는 거리 함수로 edit distance를 사용하며, 일정한 길이의 부분 문자열 내에서 각 문자가 나타난 개수를 이용해 작은 인덱스를 만든다. 위 방법을 이 논문이 대상으로 하는 wildcard문제나 k-mismatch 문제에 적용할 수는 있지만, 여러 번 반복을 통해서만 답을 찾을 수 있기 때문에 다소 비효율적이다.

3. 윈도우의 각 문자 개수를 이용한 인덱스 (Basic Signature Index (이하 BSI))

3.1. BSI 만들기

정의 1. 용어정의

T	텍스트 문자열 (DNA 데이터베이스)
T_r	텍스트 T의 r의 위치에서 윈도우의 크기(w) 만큼의 부분 문자열 (윈도우)
P	패턴 문자열 (T 에서 찾아내야 할 문자열)
m	텍스트의 크기 (=|T|)
n	패턴의 크기(=|P|)
w	윈도우의 크기
*	모든 문자와 매치될 수 있는 문자 (wildcard)

S(s)	알파벳 $\sum=\{\alpha_1, \alpha_2, \ldots, \alpha_\sigma\}$ 로 구성된 문자열 s가 있을 때, 서명 S(s)는 $([l_1,u_1],[l_2,u_2],\ldots,[l_\sigma,u_\sigma])$으로 나타낸다. ($l_i,u_i$은 각각 α_i 문자가 윈도우에 나타날 수 있는 개수의 최소값과 최대값)
p	전체 윈도우서명에서 패턴의 서명과 일치할 확률
k	k-mismatch 에서 인정하는 최대 불일치 횟수

BSI를 만들기 위해서 T의 각 윈도우 T_r마다 서명 $S(T_r)$를 먼저 구한다. 이렇게 구한 모든 서명 $S(T_r)$이 영역 형태를 나타내기 때문에 R-tree[2]를 사용해 인덱스를 구축한다.

$S(T_r)$을 계산할 때 wildcard가 없는 경우에는 l_i,u_i는 단순히 각 문자의 개수로 표현된다. 하지만 wildcard가 있는 경우에 wildcard의 위치에는 모든 문자가 나타날 수 있으므로 각 문자의 u_i를 wildcard의 개수만큼 증가시킨다.

예를 들어 $\sum=\{A, C, G, T\}$라 할 때,

T_r =ACTGGT라면 $S(T_r)= (\overset{A}{[1,1]} , \overset{C}{[1,1]} , \overset{G}{[2,2]} , \overset{T}{[2,2]})$ 이다.

T_r =ACT*GT라면 $S(T_r)= (\overset{A}{[1,2]} , \overset{C}{[1,2]} , \overset{G}{[1,2]} , \overset{T}{[2,3]})$ 이다.

3.2. BSI에서 문자열 찾기

패턴 P가 주어지면, 패턴과 일치하는 윈도우들을 BSI로부터 찾기 위해 먼저 패턴의 서명 S(P)를 구한다. 문제의 유형에 따라 S(P)를 구하는 방법이 조금씩 다르다. Exact match문제와 wildcard 문제의 경우에는 3.1절에서 제시된 방법으로 S(P)를 구한다. K-mismatch인 경우에는 k번의 불일치가 각 문자마다 삭제나 삽입으로 나타날 수 있으므로 삭제인 경우에는 각 문자의 l_i가 k만큼 감소하고 삽입인 경우에는 각 문자의 u_i가 k만큼 증가한다.

예를 들어 $\sum=\{A, C, G, T\}$라 할 때,

k=1, P=ACTGGT라면 $S(P)= (\overset{A}{[0,2]} , \overset{C}{[0,2]} , \overset{G}{[1,3]} , \overset{T}{[1,3]})$ 이다.

패턴의 서명 S(P)는 공간 상의 영역으로 표현된다. 윈도우 T_r 이 패턴 P를 만족한다면 $S(T_r)$을 표현하는 영역과 S(P)를 표현하는 영역이 반드시 중첩(overlap)한다. 그러므로, 다음 단계에서는 S(P)와 중첩이 있는 윈도우들을 인덱스 검색을 통하여 구한다. 최종 단계에서는 이전 단계에서 얻어진 윈도우의 실제 문자열을 디스크로부터 읽어서 정말로 패턴과 일치하는 가를 검증한다.

3.3. 패턴의 크기가 윈도우 크기와 다른 경우

일반적으로 패턴의 크기가 항상 윈도우 크기와 일치하지 않는다. 따라서 윈도우의 크기를 일반적인 패턴의 크기보다는 작게 설정해야 한다. 만약 패턴의 크기가 윈도우 크기보다 큰 경우에는 패턴을 몇 개의 조각으로 나눈 다음에 각 조각마다의 결과값을 찾아 그 결과들을 결합해서 답을 찾는다. 이때 패턴을 나누는 방법은 아래 그림과 같다. 패턴의 처음부터 윈도우 크기만큼 자르고, 마지막 부분은 윈도우 크기보다 작기 때문에 패턴의 끝에 맞춰 앞쪽으로 패턴의 크기만큼 자른다.

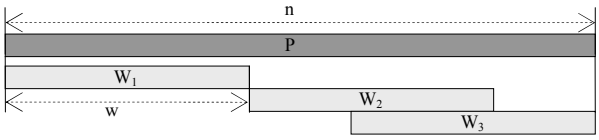

그림 2. 윈도우크기보다 큰 패턴을 분리하는 방법

3.4. BSI의 문제점

위의 인덱스 구조는 2가지 측면에서 문제점을 가지고 있다. 일단 크기가 서픽스 트리보다 줄기는 했지만 아직도 실제 텍스트보다 큰 인덱스 구조이다. 두 번째 문제점은 윈도우 서명의 분포 때문에 발생한다. 확률적으로 $[l_i,u_i]$의 분포는 평균점 근처에 많이 분포한다. 그 결과 패턴의 서명이 평균점 근처인 경우와 평균점하고 먼 경우에는 서명이 중첩될 확률 p가 큰 차이를 보이게 된다. 최악의 경우에는 p가 거의 1에 가까워져서 brute-force의 방법과 같은 시간복잡도를 보이게 된다.

4. 인덱스 크기를 줄이기 위한 방법

인덱스 크기를 줄이기 위해 윈도우 서명 몇 개를 하나의 서명으로 통합하는 방법을 이용할 수 있다. 위의 BSI의 경우에 이전 윈도우의 서명과 현재의 윈도우의 서명은 공간상에서 가까운 위치에 존재하기[4] 때문에 연속된 c개의 윈도우 서명을 하나로 묶어서 하나의 MBR(Minimum Bound Rectangle)[4]로 만든다. 이렇게 함으로써 인덱스 크기를 1/c로 줄일 수 있다. 이때 c를 box capacity로 정의한다.

5. 가중치를 갖는 문자의 개수를 이용하는 인덱스 구조 (Weighted Signature Index (이하 WSI))

BSI의 문제점은 윈도우 영역 안에 있는 문자들의 정보 중에서 위치에 따른 정보는 이용하지 않는다는 것이다. 그래서 확률적인 영향으로 전체영역의 중간에 윈도우의 서명이 집중된다. 이것을 해결하기 위해 각 문자마다 위치에 대한 정보를 포함함으로써 서명의 분포를 분산시킬 수 있다.

5.1. 위치에 따른 가중치를 이용할 때 서명 값의 변화

정의 2. 용어정의 2

w(i)	윈도우에서 i 번째 문자의 가중치
Sw(s)	알파벳 $\sum=\{\alpha_1, \alpha_2, \ldots, \alpha_\sigma\}$ 로 구성된 문자열 s가 있을 때, 서명 Sw(s)는 $([l_1,u_1],[l_2,u_2], \ldots,[l_\sigma,u_\sigma])$으로 나타낸다. ($l_i,u_i$은 각각 α_i 문자가 윈도우에서 갖는 *가중치 합*의 최소값과 최대값)

예를 들어 $\sum=\{A, C, G, T\}$, w(i)=i 라고 한다면

T_r =$\overset{1}{A}\overset{2}{C}\overset{3}{T}\overset{4}{G}\overset{5}{G}\overset{6}{T}$라면 $Sw(T_r)= (\overset{A}{[1,1]} , \overset{C}{[2,2]} , \overset{G}{[9,9]} , \overset{T}{[9,9]})$ 이다.

T_r =ACT*GT라면$Sw(T_r)= (\overset{A}{[1,5]} , \overset{C}{[2,6]} , \overset{G}{[5,9]} , \overset{T}{[9,13]})$ 이다

BSI의 경우에는 문자의 개수에 대한 정보만을 이용하기 때문에 $S(T_r)$를 이용해서 인덱스를 만들었다. 하지만 WSI에서는 문자의 개수뿐 아니라 위치에 대한 정보를 포함 시킬 수 있는 방법을 이용하기 위해 $Sw(T_r)$를 이용해서 인덱스를 만든다.

이렇게 만들어진 인덱스에서 원하는 패턴을 찾는 과정은 BSI와 동일하다.

5.2. 가중치 함수 w(i)작성에 대한 효율적인 방법

w(i)를 작성함에 있어서 한가지 고려해야 할 것이 있다. w(i)=1 처럼 한 윈도우 안의 가중치 값의 최소값과 최대값의 차이가 작으면, 단순히 문자의 개수에 대한 정보가 많이 포함된다. 그러나 w(i)=i처럼 그 차이가 크면 문자의 위치에 대한 정보가 많이 포함된다. 따라서 w(i)=i처럼 차이가 큰 경우에는 각 윈도우 서명이 넓게 퍼지겠지만 k-mismatch문제의 경우에 패턴의 서명의 범위가 커지기 때문에 좋지 않은 효율을 보이게 된다. 그렇기 때문에 가중치의 최소값과 최대값의 차이를 적절하게 조절함으로써 더욱 좋은 결과를 얻을 수 있다.

6. 실 험

실험을 위해 몇 가지 제한 사항을 두었다. 첫 번째는 일반적인 컴퓨터 환경을 모의실험하기 위해 메모리 사용을 전체 데이터 크기의 10~50%인 1MB로 한정하였다. 두 번째로, 일반적인 검색 패턴의 크기가 512보다 크기 때문에[4] 윈도우 크기는 512로 고정하였다. 마지막으로 DNA데이터에 1% 정도의 wildcard가 있는 것을 고려해서 wildcard의 개수나 k-mismatch에서 k를 전체 윈도우 크기의 1%인 5개로 고정하였다. 먼저 BSI를 이용해서 box capacity

의 변화에 따른 검색시간의 변화를 측정하였다.

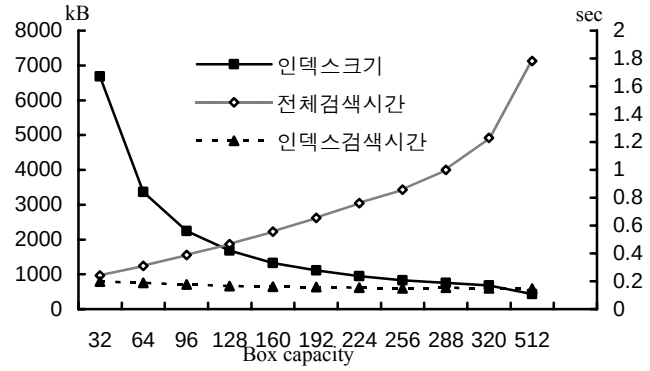

그림 3. Box capacity에 따른 검색시간 비교

위의 그림과 같이 box capacity가 커질수록 인덱스 크기가 작아져서 인덱스 검색시간은 감소한다. 하지만 검증해야 할 데이터가 많아지기 때문에 전체 검색시간은 증가한다. 따라서 공간과 시간 사이에서 적절한 box capacity를 선택 해야 할 것이다.

다음 실험들에서는 앞에서의 제한사항을 지키기 위해서 각 데이터들의 인덱스 크기가 1MB가 넘지 않으면서 최대가 되는 box capacity를 사용하였다.

다음은 논문에서 제안하고 있는 방법들의 효율성을 테스트 해보기 위해 아래와 같은 3가지 방법을 실험해 보았다.

① brute-force (이하 BF) ② BSI ③ WSI (w(i)=w+i)

텍스트 문자열 T는 2MB, 4MB, 6MB, 8MB, 10MB크기의 DNA 데이터를 이용하였고, 각 T에서 윈도우의 크기와 동일한 100개의 패턴 P를 뽑아서 질의로 사용하였다. 결과값은 각 P당 1번씩 총 100번을 테스트한 결과의 합으로 사용하였다. 그리고 논문에서 제안한 인덱스 구조를 이용해서 exact match문제와 wildcard문제, k-mismatch문제를 해결할 수 있지만 wildcard문제와 k-mismatch문제는 서명 값이 영역의 형태로 비슷하기 때문에, 서명이 점의 형태로 나타나는 exact-match문제와 영역의 형태로 나타나는 k-mismatch문제만을 비교하였다.

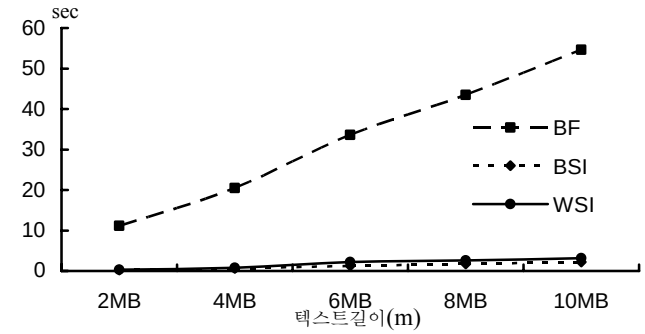

그림 4. 텍스트 크기에 따른 exact-match문제 검색시간 비교

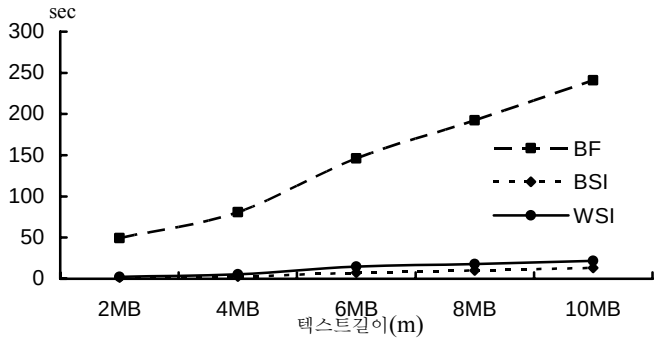

그림 5. 텍스트 크기에 따른 k-mismatch문제 검색시간 비교

그림 4,5을 보면 exact-match문제나 k-mismatch문제 모든 경우에 BSI을 이용한 방법이 BF보다는 훨씬 좋은 결과를 보이고 있다. 하지만 BSI에 비해서 WSI의 방법은 오히려 약간 나쁜 결과를 나타내고 있다. 이것은 BSI의 경우에는 인접한 윈도우가 인접한 서명 값을 갖기 때문에 MBR로 묶는 것이 효율적이지만, WSI의 경우에는 인접한 윈도우가 인접한 서명 값을 갖지 않을 수도 있으므로 MBR로 묶는 것이 비효율적이기 때문이다.

따라서 WSI의 방법의 효율성을 검증하기 위해서 MBR를 사용하지 않는 방법으로 실험을 해보았다. 그 결과 그림 6과 같이 인덱스를 줄이지 않은 상태에서는 WSI가 BSI의 40%정도의 시간으로도 결과를 찾을 수 있었다.

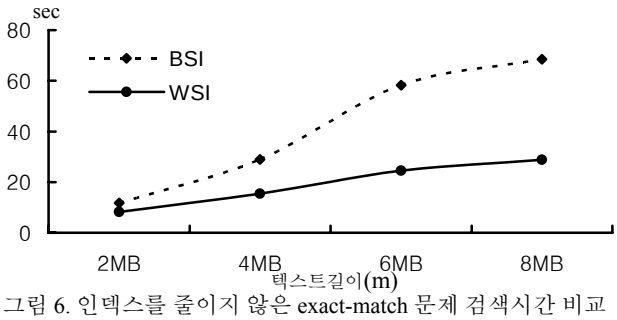

그림 6. 인덱스를 줄이지 않은 exact-match 문제 검색시간 비교

7. 결 론

이 논문에서 제안한 방법을 대용량의 문자열 데이터베이스의 하나인 DNA데이터를 이용해서 검증해보았다. 실험 결과를 통해 인덱스 크기를 실제 데이터 크기의 10%로 만들었을 경우에 BSI을 이용한 경우가 BF를 이용한 방법보다 실행 시간이 약 5%로 줄어드는 것을 알았다.

또한 MBR을 이용하지 않고 WSI를 이용하면 BSI를 이용한 경우에 비해 약 40%의 시간으로 문제를 풀 수 있었다. 이처럼 WSI를 이용하면 BF와 BSI보다 빨리 패턴을 찾을 수 있다. 하지만 이 경우에는 데이터의 크기보다 큰 인덱스가 된다. 따라서 작고 효율적인 인덱스를 만들고자 하는 논문의 목적과는 일치하지 않은 방법이다.

따라서 향후 연구에서는 WSI를 이용하면서 인덱스 크기를 줄이기 위한 방법을 찾아서 효율성을 극대화할 수 있는 방법을 찾고자 한다. 그런 방법으로, 첫 번째는 BSI와 같은 방법으로 MBR을 만들어 사용하되 가급적 MBR 영역의 크기가 커지지 않는 최적화된 w(i)를 찾는 것이다. 두 번째는 단순히 c개의 인접한 윈도우를 하나의 MBR로 이용하는 게 아니라 묶는 영역의 크기를 최소화하면서 많은 윈도우를 하나로 묶을 수 있는 방법을 찾을 것이다.

참고문헌

[1] D. Gusfield, Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, 1 edition, January 1997.

[2] A. Guttman, “R-Trees, A dynamic index structure for spatial searching”, In Proceeding of ACM SIGMOD, 1984.

[3] 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.

[4] T. Kaheci, A. K. Singh, “An Efficient Index Structure for String Databases”, In Proceeding of VLDB Conference, 2001.