압축된 인덱스 자료구조를 위한 구축 및 검색 알고리즘의 성능 분석

이분녀, 김동규
부산대학교 컴퓨터공학과

Analysis of Construction and Searching Algorithms for Compressed Index Data Structures

Bun Nyeo Lee, Dong Kyue Kim
Dept. of Computer Engineering, Pusan National University

요 약

기하급수적으로 증가하고 있는 방대한 양의 데이터를 효율적으로 저장하고, 검색하기 위한 방법으로 압축된 인덱스 자료구조(compressed index data structure)가 제안되었고 현재 활발히 연구되고 있다. 압축된 인덱스 자료구조란 데이터를 적절한 방법으로 색인화(indexing)하여 이를 압축(compression)된 자료구조로 저장하여, 데이터 검색의 성능저하 없이, 저장 공간을 줄일 수 있는 방법이다. 본 논문에서는 대표적인 방법으로 Ferragina와 Manzini가 제시한 FM-index를 다룬다. 이 방법을 구현하여 전체적인 성능에 영향을 미치는 요소들을 실험을 통해 분석하였다. 이를 통하여 각 파라미터들의 상관관계를 분석하고 이상적인 설정 값을 실험하였다.

1. 서론

웹 검색엔진(search engines), 유전체 데이터베이스(genome database) 등의 발전에 따른 데이터양의 증대는 빠른 검색을 위해 전체 텍스트에 대한 인덱스 자료구조를 필요로 하게 되었다. 이 중 접미사 배열(suffix array)과 접미사 트리(suffix tree) 등이 활발히 연구되고 있는데, 폭발적으로 증가하는 데이터의 크기가 컴퓨터 메모리와 디스크의 저장 용량보다 훨씬 빠르게 증가하고 있는 실정에서, 수집된 방대한 데이터들 속에서 사용자의 요구에 필요한 정확한 정보를 빠르게 찾아내기란 쉬운 일이 아니다. 이러한 정보 자원들 속에서 원하는 정보를 보다 빠르고 정확히 찾으면서 저장 용량을 줄일 수 있는 방법이 필요하게 되었다. 그래서 데이터의 압축(compression)과 색인화(indexing)를 결합한 방법들이 제안되고 연구되고 있는데, 그 중에서 Ferragina와 Manzini가 제안한 FM-index[7]는 $O(n)$ 비트를 가지고 $O(n)$ 시간에 적은 용량을 가지면서도 데이터 검색 성능이 저하되지 않는 압축된 색인 자료구조를 생성할 수 있는 방법이다.

* 이 논문은 2003년도 학술진흥재단의 지원에 의하여 연구되었음(KRF-2003-003-D00343).

본 논문에서는 Ferragina와 Manzini가 제안한 FM-index를 살펴보고, 이를 구현하고, 실험을 통하여 성능을 분석하고자한다.

본 논문의 구성은 다음과 같다. 먼저 2장에서는 FM-index에 대해 살펴보고, 3장에서는 구현방법에 대해 살펴보고, 4장에서는 실험을 통해 성능을 분석하고, 마지막으로 5장의 결론에서는 향후 연구 과제를 제시한다.

2. 배경 지식

FM-index(full-text in minute space)는 검색 성능의 저하가 적으면서 저장 공간을 적게 쓰는 데이터 구조로, 압축과 색인화를 조합한 형태의 데이터 구조이다. 기존의 압축 알고리즘과의 차이점은 정보의 검색이 가능한 색인을 사용할 수 있는 압축된 색인을 생성하는 방법이다. 즉, 적은 공간을 차지하는 전체 텍스트에 대한 인덱스 자료구조이다. 이 FM-index를 사용하여 압축된 파일의 일부만 참조하여 찾고자 하는 패턴의 원래 파일에서의 발생 개수를 셀 수 있고, 그 위치도 찾을 수 있다.

검색 성능을 보장하는 압축된 색인을 생성하기 위해 Ferragina와 Manzini는 Burrows와 Wheeler의 BW-Transform을 이용하였다[2-4]. 접미사 배열을 가

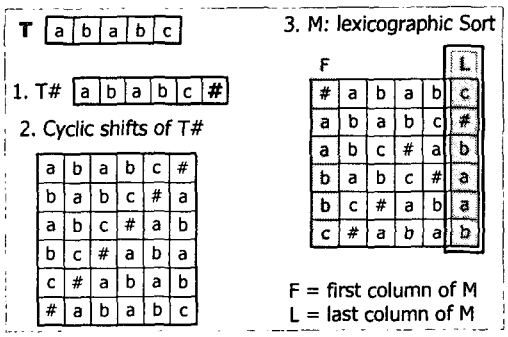
지고 이 알고리즘을 적용하면 압축된 인덱스 자료구조를 얻을 수 있다. 그 결과 색인이 선택적이기 때문에 opportunistic 자료구조라고도 한다. 이 BW-Transform을 통해 중요하지 않은 부분은 공간을 줄임으로써 입력 데이터를 압축할 수 있는데, 텍스트 T를 색인화 하려면 각 문자마다 $O(H_k(T))+o(1)$ 비트 (단, k는 $k \geq 0$인 고정상수)의 저장 공간이 필요하다 [7]. 이때, $H_k(T)$는 텍스트를 최대한 압축했을 때의 엔트로피(entropy)를 나타낸다. 이렇게 생성된 색인을 통해 임의의 패턴 $P[1,p]$에 대해 $O(p)$ 시간 안에 텍스트 내에서 패턴의 발생 개수를 알 수 있으며 각 발생에 대해 $O(\log^{\epsilon} u)$ 시간 (단, ϵ은 $\epsilon > 0$인 고정상수) 안에 열거할 수 있다.

압축된 색인 정보를 얻기 위해 Ferragina와 Manzini가 이용한 BW-Transform을 간단히 살펴본다. BW-Transform 알고리즘은 텍스트 T와 접미사 배열을 입력으로 받아서 압축된 접미사 배열을 생성한다[5]. 알고리즘은 다음과 같다:

알파벳 Σ로 된 텍스트를 $T[1,u]$라고 하면,

(1) 텍스트 끝에 모든 Σ보다 선행하는 #을 붙인다.
(2) T# 문자열들을 순환 쉬프트 시킨 후 각 행을 사전 순서로 정렬하여 개념 매트릭스 M 생성.
(3) M에서 가장 마지막 열을 취하여 L 배열 생성.
(4) 압축 알고리즘[8]을 적용, L 배열을 압축한다.

[그림 1]은 BW-Transform 알고리즘을 나타낸 것이다.



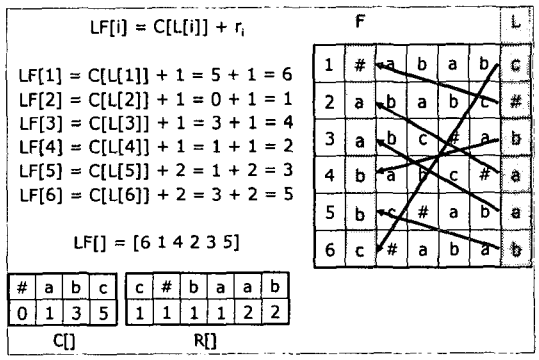
[그림 1] BW-Transform을 이용한 L의 생성

BW-Transform에서 개념 매트릭스 M의 각 행을 정렬할 때 T의 접미사들을 정렬한다. 이것은 개념 매트릭스 M과 접미사 배열과는 밀접한 관계가 있음을 말해준다. 이 관계는 FM-index 자료구조의 설계에서 중심 개념이 된다. 개념 매트릭스 M의 특성을 살펴보기 위해 다음과 같이 정의한다:

- $c \in \Sigma$에 대해, T에서 c보다 작은 알파벳의 총 발생 개수를 저장한 배열을 $C[c]$라고 한다.
- $c \in \Sigma$에 대해, 변환된 L의 접두사 $L[1,k]$에서의 c의 발생 개수를 $Occ(c,k)$로 나타낸다.

개념 매트릭스 M은 다음과 같은 특성이 있다:

a. 주어진 M의 i번째 행에 대해, 마지막 열의 문자 $L[i]$는 원래 텍스트 T에서, M의 첫 번째 열의 문자 $F[i]$를 선행한다.
b. $C[L[i]] + Occ(L[i], i) = LF(i)$라고 하자. $L[i]$와 대응되는 M의 첫 번째 열 F는 $LF[i]$에 위치한다. 이것을 LF-mapping 이라고 한다. [그림 2]에서 LF-mapping의 예를 보여주고 있다.
c. $T[k-1] = L[LF(i)]$이면 $T[k]$는 L의 i번째 문자이다.



[그림 2] LF-mapping의 예

FM-index는 압축된 변환 배열 L과 임의의 패턴 검색에 쓰일 약간의 보조 정보(auxiliary information)인 Aux로 구성된다. 이렇게 생성된 압축된 인덱스 배열인 FM-index가 제공하는 검색은 *count*와 *locate*가 있다.

*count*는 텍스트 T에서 특정 패턴 P의 발생 개수를 결정하는 것이다. *count*를 위해 $P[i,p]$로 시작하는 M의 첫 행을 sp라고 하고, $P[i,p]$로 시작하는 마지막 행을 ep라고 할 때 $(ep-sp+1)$이 T 안에 존재하는 P의 총 발생 개수이다(단, $1 \leq i \leq p$). 이 *count*의 계산은 최대 패턴의 길이 $|P|$ 만큼의 시간, $O(p)$시간이 걸린다.

*locate*는 텍스트 T에 존재하는 패턴의 위치를 찾아주는 함수이다. 이 함수는 입력으로 M 매트릭스의

행의 색인 i를 입력으로 받아 $M[i]$에 대응되는 접미사의 T에서의 시작 위치, $pos(i)$를 반환해 준다. M의 일부 행을 적절히 마킹하여 T에서의 위치를 저장했다가, i가 마킹되어 있으면 바로 $pos(i)$를 사용할 수 있는 구조로 이용한다. 이때, 만약 i가 마킹되어 있지 않으면 $T[pos(i)-1, u]$ 접미사로 대응되는 행 i_1을 찾기 위해 LF-mapping과 C 배열을 이용하여, $pos(i_v)$로 마킹된 행 i_v에 도달할 때까지 v번 반복한 후 $pos(i) = pos(i_v) + v$ 값으로 설정되며, 이 모듈을 위해서는 $O(\log^c u)$만큼 시간이 소요된다.

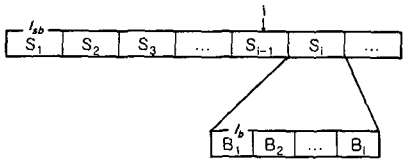
·이상에서 살펴본 바에 의하면, Occ에 대한 효율적 구현과 검색을 위한 적절한 마킹 전략이 중요한 요소가 된다.

3. 구현방법

이 장에서 다루게 될 FM-index의 구현은 논문 [7, 13]에서 소개한 방법에 기초를 두고 있다. 본 논문에서는 FM-index의 구현을 L 생성과 *count*와 *locate*에 의한 검색의 두 부분으로 나누어 구현하였다. 전체 알고리즘의 성능을 좌우하는 중요 변수를 중심으로 구현에 대해 자세히 살펴본다.

3.1 Occ의 구현

Occ를 계산하기 위해 변환된 L을 크기 l_{sb}의 super buckets으로 나눈다. 각 superbucket은 다시 크기 l_b의 bucket으로 나눈다. 각 superbucket에는 배열 L의 시작부터 각 문자의 발생 개수를 담고 있는 테이블을 저장한다. 즉, superbucket S_i는 $S_1, S_2, ..., S_{i-1}$까지의 c의 발생 개수를 담고 있다. 마찬가지로, 각 bucket은 superbucket의 시작부터 모든 문자들에 대한 발생 개수를 담고 있다. 이 보조 정보를 이용하여, 쉽게 L부터 그 bucket의 시작 위치까지의 각 문자의 발생 개수를 계산할 수 있다. 아래 [그림 3]은 위의 설명과 같이 superbucket과 bucket으로 나눈 모양이다.



[그림 3] Occ의 계산을 위한 구조

· superbucket : 이전 superbucket에서의 모든 문자에 대한 발생 개수를 각 superbucket에 저장한다.
· bucket directory : FM-index의 body에서 압축된 각 bucket의 시작 위치를 저장한다.
· body : 각 bucket의 압축된 정보를 저장한다.

$Occ(c,k)$를 계산하기 위해 위의 데이터 구조를 준비하고 먼저 *locate* $L[k]$를 포함하는 bucket B_i의 시작 위치를 bucket directory를 이용하여 처리한다. 그런 다음 B_i의 압축을 해제하고, bucket의 시작 위치로부터 $L[k]$까지 c의 발생 개수를 카운트한다. 그런 다음 bucket header에서 superbucket의 시작 위치부터 c의 발생 개수를 얻는다. 마지막으로 superbucket 부분에서 L의 시작부터 c의 발생 개수를 얻는다. 여기서 superbucket과 bucket의 크기에 따라 성능이 달라질 수 있다.

3.2 저장 효율성을 위한 Bitmap의 처리

좀 더 빠르게 $Occ(c,k)$를 계산하기 위한 방법으로 한 가지 더 추가할 사항으로는 각 bucket에서 저장된 발생 문자에 대해 bitmap으로 처리하여 저장하는 것이다. 이렇게 함으로써 어떤 문자가 어떤 bucket에 있는 지를 좀 더 빨리 검색할 수 있고, 따라서 $Occ(c,k)$를 빠르게 계산할 수 있다.

3.3 개념 매트릭스 M의 마킹 처리

마지막으로 개념 매트릭스 M의 마킹 정보의 크기는 Ferragina와 Manzini의 논문 [7]과 [9]번의 내용에 의거, 입력의 2%를 저장하여 처리하였다.

4. 실험 결과 및 성능 분석

이번 장에서는 성능에 영향을 줄 수 있는 요소를 실험을 통해 살펴본다. 실험 환경은 Pentium-IV 2.66GHz, IDE hard disk, 512Mb, XP 환경에서 실험했으며, Visusal C++ v6.0으로 컴파일 하였다.

인덱스 자료를 압축하기 위한 4가지 압축기법들의 성능을 비교하였고, 전체 검색 성능과 저장 공간에 영향을 미치는 중요한 요소로 superbucket의 크기를 16[Kb]부터 1024[Kb]까지 다양하게 조정하여 성능과의 상관관계를 규명하였다. 아래 표들은 실험 결과에 따른 성능과 파라미터의 상관관계를 분석한 결과이다.

· Bucket의 압축 방법에 따른 실험 결과

코딩방법	Arithmetic	Hierarchic	Unary	Huffman
압축률[%]	36.51	56.11	54.02	33.92
bit수/char	2.92	4.49	4.32	2.71
생성시간	0.92	0.86	0.78	0.94

위 실험은 논문 [7]에 소개된 압축 기법들을 적용하

여 실험한 결과이다. 이 결과를 통해 알 수 있는 것은 huffman 코딩과 arithmetic 코딩이 압축 및 속도 면에서 상대적으로 결과가 좋았다.

· Superbucket의 크기에 따른 실험 결과

l_{sb}[Kb]	16	32	128	1024
압축률[%]	33.07	33.01	33.17	33.57
bit수/char	2.65	2.64	2.65	2.69
생성시간[ms]	3.70	3.75	3.72	3.69

Huffman 코딩, mito를 이용한 실험으로 이 실험 결과를 통해 superbucket의 크기는 전체 성능에 큰 영향을 미치지 않는다는 사실을 알 수 있다.

· 입력 데이터별 실험 결과

Data	Size[byte]		압축률	bit수	생성시간
	前	後	[%]	/char	[ms]
alu	80,506	21,597	26.83	2.15	0.05
ecoli	4,662,239	1,566,537	33.06	2.69	5.49
mito	3,164,247	1,046,491	33.07	2.65	3.97
pdb	226,619	40,681	17.95	1.44	0.20
vector	3,613,945	466,273	12.90	1.03	8.64
yeast	12,155,026	4,190,858	34.48	2.76	15.80

이 실험은 입력 데이터의 크기와 종류를 달리하여 실험을 한 결과이다. $|\Sigma|$=4, huffman 코딩, l_{sb}=16[Kb], l_b=1[Kb]로 설정하여 실험한 결과, 평균 압축률이 26.38[%] 정도 나왔다. 또한 이러한 압축률에도 불구하고 다음 결과를 보면 검색 성능에는 큰 저하가 없다는 것을 확인할 수 있다.

· 패턴 크기별 검색 시간

Size[M]	1	2.5	3	4	5	10
count[ms]	0	0.01	0.02	0.01	0	0
locate[ms]	7.53	9.01	7.0	7.6	7.9	73.87

5. 결론

Burrows-Wheeler의 압축 알고리즘을 이용하여 압축된 접미사 배열을 생성하는 Ferragina와 Manzini의 FM-index는 기존의 압축 알고리즘과는 달리 자료를 검색할 수 있도록 색인을 제공한다. 그러면서도 평균적으로 27%의 압축률로 데이터를 압축하여 저장함으로써 검색 성능의 큰 저하 없이 저장 공간을 줄일 수 있다는 사실을 실험을 통해 확인해 보았다. 3장에서 살펴본 실험 결과에 영향을 미치는 중요 요소인 Occ를 $O(1)$시간 안에 계산하기 위한 효율적 구현방법과 충분한 검색 성능을 위해 보조 정보의 저장량을 결정하는 superbucket의 크기가 전체 성능에 크게 영향을 미치지 않는다는 사실을 확인하였다.

그 외에도 개념 매트릭스의 마킹량을 본 논문의 실험에서는 입력의 2%로 고정하여 실험을 하였기에 마킹량과 성능의 직접적인 상관관계를 밝히지는 못했다. 차후, 이러한 관계를 살펴봄으로써 가장 이상적인 마킹량에 대해 살펴보는 것도 좋을 것이다.

[참고문헌]

[1] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262-272, 1976.

[2] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[3] P. Fenwick. The Burrows-Wheeler transform for block sorting text compression: principles and improvements. *The Computer Journal*, 39(9):731-740, 1996.

[4] G. Manzini. An analysis of the Burrows-Wheeler transform. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, 669-677. Full version in Tech. Rep. B4-99-13, IMC-CNR, 1999. http://www.imc.pi.cnr.it/~manzini/tr-99-13/.

[5] J. I. Munro. Succinct data structures. In *Proceedings of the 19th Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag LNCS n. 1738, 1999.

[6] J. I. Munro. Tables. In *Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS '96)*, pages 37-42. Springer-Verlag LNCS n. 1738, 1999.

[7] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st IEEE Symposium on Foundations of Computer Science*, 2000.

[8] J. Bentley, D. Sleator, R. Tarjan, and V. Wei. A locally adaptive compression scheme. *Communication of the ACM*, 29(4):320-330, 1986.

[9] P. Ferragina and G. Manzini. An experimental study of an compressed indexing.