문서 검색 시스템을 위한 도치 색인 파일의 압축 저장 기법 개선*

이준영, 김민정, 권혁철
부산대학교 전자계산학과

An Improved Bit Vector Compression Method for a Document Retrieval System

Jun-Young Lee, Min-Jung Kim, Hyuk-Chul Kwon
Dept. Of Computer Science, Pusan National University

abstract

문서 검색 시스템의 도치 색인 파일은 저장 공간과 검색 시간을 줄이기 위해 색인어 사전과 문서번호를 위한 이진 도치 파일 또는 비트벡터로 구성할 수 있다. 비트벡터는 1의 값을 가지는 비트만 저장함으로써 압축이 가능하나, Bit tree 압축 방법은 block의 크기가 어떻게 결정되느냐에 따라 압축률이 달라진다. 본 논문에서는 비트벡터에 나타나는 1의 값을 가지는 비트의 개수에 의해 bit tree 압축에 대한 최적의 block 크기를 결정하는 방법을 했다. 최적의 block의 크기는 {전체 비트벡터의 크기 / 1의 개수}보다 작거나 같은 최대의 2의 승수이다. 또한 block내의 1의 값을 가지는 비트의 위치에 의해 상대 위치값을 계산해서 block의 오른쪽 반에 나타나는 비트를 더 압축할 수 있는 기법을 구현한다. 본 논문에서 구현한 압축 기법은 Run Length를 이용한 방법에 비해서 13.65%, 기존의 Bit Tree 방법에 비해서 1.88%의 압축률을 개선했다.

1. 서론

문서 검색 시스템의 기본적인 도치 색인 파일은 색인어와 이들 색인어를 포함하는 문서 번호들로 구성된다. 색인어에 따라 해당 문서들의 수가 다르므로 문서번호 리스트만 따로 저장하면 저장공간을 효율적으로 사용할 수 있다. 특히 방대한 양의 문서번호 리스트를 그대로 저장하는 대신에 0과 1을 값으로 가지는 이진 도치 파일 형태인 비트벡터로 구성하면 저장공간을 줄일 수 있다. 대용량의 문서 검색 시스템에서의 비트벡터는 1의 값을 가지는 비트가 0의 값을 가지는 비트에 비해 sparse하다[1][2]. 따라서 sparse하다는 특징을 이용하여 압축률을 더 높일 수 있다.

비트벡터에서 비트 값이 1로 나타나는 경우는 그 위치에 해당하는 번호의 문서에 해당 색인어가 있음을 나타내고, 0으로 나타나는 비트는 해당 색인어가 없음을 의미한다. 따라서 전체 문헌수가 N이라 하면, 비트벡터는 모두 N개의 0과 1로서 구성된다. 즉 n번째 문헌에 해당 색인어가 존재하면, 비트벡터의 n번째 bit는 1로 표현된다. 그러나 비트벡터 기법의 경우 α건의 자료에 대해 색인어가 β개이면, 색인정보를 저장하기 위해 소요되는 기억용량은 $\alpha * \beta$ bit가 된다. 따라서 200만 건의 자료가 50만 색인어에 의해 색인될 경우에는 200만 * 50만 bit의 기억용량이 필요하고, 이는 1,000G bit(125G byte)해 해당되는 양이다. 즉, 자료의 양이 증가함에 따라, 저장 용량의 크기는 기하급수적으로 늘어나고, 저장된 정보의 검색 시간도 이에 비례해 늘어나게 된다. 따라서, 비트벡터 기법은 문서가 수백만 건에 이르는 대용량 문서 검색 시스템에서는 저장 및 검색의 효율성을 보장할 수 없다. 따라서 대용량 문서

* 본 연구는 '93년도 한국 과학 재단 목적 기초 협력 연구 과제 연구비에 의해 연구되었음.

검색 시스템에서 비트벡터의 압축은 필수적이다.

본 논문에서는 비트벡터 압축 기술로는 Run Length와 Bit Tree 방법을 실험했다. 특히, Bit Tree를 이용한 방법 중에서 prefix omission을 이용한 방법[1]의 최적의 압축을 할 수 있는 block 크기를 결정하는 방법을 제안한다. 또한 최적의 block 크기를 적용하고 block 내의 1의 분포를 이용해서 기존의 prefix omission을 이용한 방법을 개선시킨 방법을 구현하였다.

2. 기존의 압축 기법

비트벡터 압축 방법으로 Run Length 와 Bit Tree 방법, huffman 방법 등이 있다. 특히 대용량의 문서 검색 시스템에서 비트벡터는 sparse하다는 특징을 이용해서 압축 효과를 더 높일 수 있다. 본 논문에서는 sparse한 비트벡터에 대해 압축률이 좋은[1][2] Run Length와 Bit Tree를 이용한 방법들의 특징을 살펴본다.

2.1 Run Length 압축 기법

Run Length를 이용한 방법은 비트벡터에서 1의 값으로 나타나는 bit 사이의 간격 값을 표시한다. Run Length를 이용한 방법으로 Elias 감마(γ)코드[3]와 델타(δ)코드[3]에 대한 몇 가지 예를 [표1]에서 나타내었다. 감마(γ)코드는 $\lfloor \log n \rfloor$비트의 0와 $1+\lfloor \log n \rfloor$만큼의 비트로 n을 지장한다. 델타 코드는 감마코드와 유사하며 $1 + \lfloor \log(1+\log n) \rfloor + \lfloor \log n \rfloor$의 메모리에 저장할 수 있다.

문서번호	앞문서와의 거리	감마코드	델타코드
20	20	00010101	001010101
30	10	0001010	00100010
65	35	00000100011	0011000011
66	1	1	1

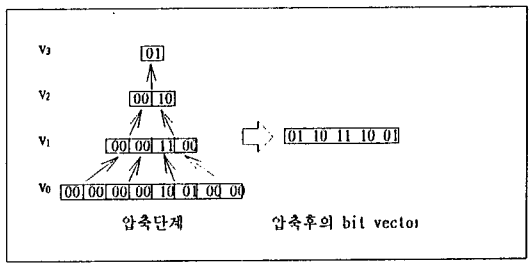
[표1] 감마코드와 델타코드의 예

두 코드를 비교해 보면 15보다 작은 값에 대해서는 감마코드가 더 효율적이나 15이상의 값에 대해서는 델타코드가 더 짧게 나타난다[2]. 뿐만 아니라 코드의 앞부분에 0으로 나타나는 부분이 감마코드에서는 $\lfloor \log n \rfloor$비트이지만 델타코드에서는 $\lfloor \log(1+\log n) \rfloor$비트로 줄임으로써 큰 수를 코드로 만들 경우에 증가율이 더 작다. 그러므로 비트벡터가 크고 sparse한 대용량의 문서 검색 시스템일 경우에 감마코드에 비해 압축효과가 크다.

2.2 Bit Tree

Bit Tree를 이용한 압축 방법은 비트벡터를 같은 크기의 block으로 구분한다. 다음 한 block 내에 비트 값이 1이 있는지를 조사해 본다. 한 block내의 비트가 모두 0이라면 그 block 값은 0이 되고 한 비트라도 1의 값을 가질 경우 block값은 1이 되고 그 위치가 지장된다.

계층적 Bit Tree 압축방법은 레벨 i번째의 비트벡터 v_i를 같은 크기의 block으로 나누고 non-zero block만으로 레벨 $i+1$번째의 비트벡터 v_{i+1}을 만든다. 이런 과정을 반복한다. 예를 들어 비트벡터 '0000000010010000'을 block의 크기를 2로 해서 계층적 Bit Tree 압축방법을 적용하면 [그림1]과 같다.



[그림1] 계층적 Bit Tree 압축 방법

계층마다 같은 크기의 block으로 나눈다면 전체 비트벡터 N에 대해서 이론적으로 $\lceil \log_{|block|} N \rceil$계층까지의 단계를 거치지만 어느 계층까지 압축을 해야만 최적이 되는지는 비트벡터의 분포에 따라서 달라진다. 계층에 따라서 block의 크기를 가변적으로 정할 수도 있다. 그러나 block의 크기를 어떻게 결정해야 하는가의 문제가 있다.

prefix omission을 이용한 Bit Tree 압축 방법[1]은 여러 단계를 두지 않고 일정한 block 크기를 결정하고 그 block 값을 나타내는 한 단계의 레벨을 두며 $\lfloor \log |block| \rfloor$비트에 그 block내에서의 위치 값을 저장하는 방법이다. 같은 block내에 1의 값을 가지는 비트가 복수개 나타날 수도 있으므로 그 block내에서 마지막으로 나타나는지를 나타내는 한 비트가 필요하다.

2.3 Run Length와 Bit Tree의 비교

Run Length를 이용한 델타(δ)code와 prefix omission

을 이용한 Bit Tree 압축 방법을 임의의 데이터를 generation해서 실험해 본 결과는 [표2]와 같다. 본 실험 결과는 비트벡터를 임의로 generation한 경우[random]와 특정 부분에 집중적으로 1이 나타나는 경우[burst]의 압축률을 나타낸다. 각각의 경우마다 10개씩의 실험 데이터를 사용했고 압축률은 평균값으로 계산했다. Bit Tree를 이용한 방법의 block 크기는 본 논문에서 제시한 방법에 의해 결정했다.

Run Length를 이용한 방법은 구현이 간단하고 비트벡터 내의 0과 1의 값을 가지는 비트의 분포에 큰 영향을 받았다. 즉 1의 값을 가지는 비트가 특정 부분에 burst한 분포를 가지는 비트벡터의 압축률은 Bit Tree를 이용한 방법보다 63% 이상 좋게 나타났다. Bit Tree를 이용한 방법의 경우 압축률은 0과 1의 값을 가지는 비트의 분포에 무관하며, 전체 비트벡터 내에서 나타나는 1의 값을 가지는 비트의 수에만 영향을 받는 것으로 나타났다.

확률	run length(%)		기존의 bit tree(%)	
	random	burst	random	burst
1/10,000	0.1916	0.0384	0.1538	0.1512
1/1024	1.5178	0.2661	1.1726	1.1114
1/128	8.9868	2.0669	7.0308	6.5560
1/4	110.9368	66.1600	100.0000	100.0000

[표2] Run Length를 이용한 방법과 Bit Tree를 이용한 압축률의 비교

3. Bit Tree를 이용한 방법에서 block 크기 결정

prefix omission을 이용한 Bit Tree 압축 방법은 block의 크기에 의해 압축률이 다르게 나타났다. 기존의 prefix omission을 이용한 Bit Tree 압축 방법은 비트벡터 내의 0과 1의 값을 가지는 비트 값들의 분포에 영향을 받지 않고 전체 비트벡터의 크기에 대해서 1의 값을 가지는 비트의 수, 즉 1이 나타난 빈도수 의해서만 영향을 받는다. 따라서 이 확률을 이용해서 최적의 압축을 위한 block 크기를 결정할 수 있다. block의 크기는 구현상 2의 승수로 제한한다. 최적의 block 크기는 다음과 같이 구해진다.

l_0를 비트벡터의 총 bit수, p를 비트벡터에서 1의 값을 가지는 bit가 나타날 확률, r_0를 block의 크기, N을 1의 값을 가지는 비트의 수 그리고 k_0를 block의 개수라고 할 때

$l_0 = 2^x$, $p = 2^{-y}$, $r_0 = 2^c$,
$N = l_0 * p = 2^{x-y}$, $k_0 = l_0 / r_0 = 2^{x-c}$

로 두면 압축후의 크기는 다음과 같다.

$$V_{size} = k_0 + (c+1)N = 2^{x-c} + (c+1)2^{x-y} \quad [식3.1]$$

[식3.1]을 c의 함수 f로 나타내면 다음과 같다.

$$f(c) = 2^{x-c} + (c+1)2^{x-y} \quad [식3.2]$$

[식3.2]에서 V_{size}가 최소값을 가지도록 하는 c는 다음과 같다.

$$c = y + log_2(ln2) \fallingdotseq y - 0.53 \quad [식3.3]$$

[식3.3]은 $c = \langle y-0.53 \rangle$ (단, $\langle x \rangle = n$, $n-0.5 \le x < n+0.5$) 와 같이 정의할 수 있다. [식3.3]에서 y가 정수이면 c = y, 즉 block의 크기는 2^y이 된다. y가 정수가 아니라면 $c+0.03 \le y < c+1.03$ 이므로 $0 < y - \lfloor y \rfloor < 0.03$ (단, int(y)는 실수 y의 정수값)의 경우에는 c = y - 1, 즉 block의 크기는 2^{y-1}이 된다. 그 외의 y의 값에 대해서는 c = y가 된다. 따라서 이론적인 최적의 압축 조건은 block의 크기가 y-0.53일 때이며 다음과 같다.

$$f(y-0.53) = 2^{x-y+0.53} + (y+0.47)*2^{x-y} = 2^{x-y}(y+1.19)$$

이 때의 압축률은 $\frac{y+1.91}{2^y}$ 즉, p(y+1.91)로 계산할 수 있다. 그러나 구현할 때는 컴퓨터 구조의 특성상 y는 정수로 제한해야 하므로 0.03의 값은 무시했다. 따라서 최적의 압축을 위한 block의 크기는 [표3]과 같이 구한다.

1이 나타날 확률 : $p = \frac{1}{2^y} = \frac{\text{1의비트개수}}{\text{전체비트벡터의크기}}$
block 크기 : $2^c = \frac{1}{2^{\lfloor y \rfloor}}$

[표3] 최적의 block 크기

또한 압축률은 1보다 작아야 한다.

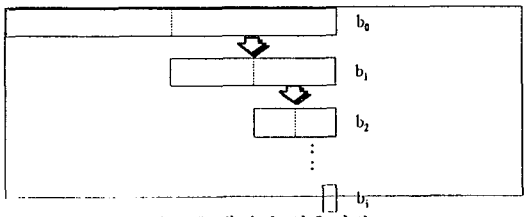
$$\frac{y+1.91}{2^y} < 1 \quad [식 3.4]$$

[식3.4]를 만족하는 정수 y의 최소값은 2이다. 즉 y가 2보다 작으면 압축률이 1보다 커진다. 따라서 확률 p가 $\frac{1}{4}$보다 작아야 압축 효과가 있다.

4. 개선된 비트벡터 압축 기법

prefix omission을 이용한 Bit Tree를 이용한 방법에서는 0과 1의 값을 가지는 비트의 분포와 압축률은 서로 독

럽적이다. 그러나 block내에서 복수개의 1의 값을 가지는 비트가 나타날 경우 1의 값을 가지는 비트간의 상대적인 거리를 계산해서 가변적인 크기로 1의 값을 가지는 비트의 위치 값을 저장할 수 있다. 즉, 한 block 내에서 가운데 비트 또는 그 이후에 어떤 한 비트가 1이면 뒤에 나타나는 1로 세트된 비트의 위치 값을 저장하는데 한 비트를 줄일 수 있다. [표4]에서 $b_i = b_{i-1}/2$이며 b_0는 전체 비트백터에서 구해진 최적 크기의 block이다.



[표4] 개선된 압축기법

이 방법은 첫째 b_0내의 가운데 또는 오른쪽 위치에 1의 값을 가지는 비트가 있다면, 그 이후에 나타나는 1의 값을 가지는 비트는 위치 값을 저장하는데 한 비트를 줄일 수 있다. 같은 방법으로 b_i위치에 나타나는 1의 값을 가지는 비트는 $\lfloor \log|b_0| \rfloor - i$의 기억 공간에 저장할 수 있다. 둘째, b_0의 마지막 위치의 비트를 바로 앞에 나타나는 1의 값을 가지는 비트의 end flag를 이용해서 표현할 수 있다. 세째, b_0의 마지막 위치에는 end flag가 필요 없다.

```
1로 세트된 비트;
if (b0에서 최초로 나온 1이면)
    현재 block을 1로 세트;
else
    바로 이전의 end flag를 0으로 reset;
b0내에서의 상대적인 위치;
if (그 b0내에서 마지막 위치의 비트이면)
   if (만약 바로 앞의 비트가 1이었다면)
      skip;
   else
      bi에서의 위치값 계산;
      비트백터에 [log|b0|]-i만큼의 저장공간에 세트;
else
   bi에서의 위치값 계산;
   비트백터에 [log|b0|]-i만큼의 저장공간에 세트;
   end flag 세트;
if (bi에서 bi/2이상의 위치의 비트였다면)
   i 증가;
```

[표5]는 비트백터에서 block의 크기가 2^3일 때 몇가지 경우에 대한 예이다. 기존의 prefix omission을 이용한 방법과 비교하였다.

비트백터	기존의 방법	개선한 방법
00001000	1 1001	1 1001
10000010	1 0000 1101	1 0000 1101
00000001	1 1111	1 111
00001010	1 1000 1101	1 1000 011
00010111	1 0110 1010 1100 1111	1 0110 010 00

*밑줄은 end flag 비트(1 : end, 0 : continue)

[표5] prefix omission을 이용한 방법과 개선된 방법의 예

5. 실험 및 결과분석

Run Length를 이용한 델타코드, 기존의 prefix omission을 이용한 Bit Tree와 기존의 prefix omission을 이용한 bit tree를 압축기법을 개선한 압축방법의 압축률을 비교, 분석한다.

[표6]은 1,000,000 비트 크기의 비트백터에서 임의의 위치에 4가지의 확률로 1이 나타나는 비트를 generation한 데이터에 대한 실험결과이다. 본 연구에서 구현한 기법은 기존의 prefix omission을 이용한 Bit Tree를 이용한 방법에 비해 압축률이 높게 나타났으며, 최악의 경우 동일한 결과가 나타났다.

확률	run length (%)	기존의 bit tree(%)	개선된 방법(%)
1/10,000	0.1916	0.1538	0.1530
1/1024	1.5178	1.1726	1.1650
1/128	8.9868	7.0308	6.9015
1/4	110.9368	100.0000	85.9335

[표6] 메뉴얼 데이터의 실험 결과

또한 압축된 비트백터내에서 하나의 위치 값을 탐색하는데 걸리는 시간은 SUN sparc 10에서 UNIX time명령어로 0.0u, 0.0s의 속도를 냈다. 그리고 250,000개의 1의 값을 가지는 비트가 있는 비트백터를 압축하는데 걸리는 시간은 Run Length는 11.8u, 0.2s, 기존의 prefix omission을 이용한 Bit Tree방법은 11.9u, 0.2s, 개선된 방법은 12.1u, 0.2s이었다.

본 연구에서는 앞에서와 같이 메뉴얼 데이터 이외에 실제 신문기사를 색인하여 실험하였다[표7]. 색인은 부산대에서 개발한 자동 색인기[6]를 사용했다. 색인기는 형태소 분석기에 의해 명사를 추출한다. 본 논문에서는 단일명사와 복합명사를 색인어로 사용한다. 실험결과는 [표7]과 같이 나타났다.

	bit vector (byte)	run length (byte)	기존의 bit tree (byte)	개선된 방법 (byte)
93년 12월	16,148,129	362,736	321,403	314,454
		2.8411%	2.4038%	2.3498%
94년 7월	15,998,394	305,473	266,995	262,744
		2.7008%	2.3345%	2.2390%

[표7] 실 데이터의 실험 결과

실 데이터에서도 generation 데이터를 실험했을때와 마찬가지로 개선된 압축률을 나타냈다. 실험 결과 개선한 기법은 Run Length를 이용한 방법보다는 13.65%, 기존의 prefix omission을 이용한 방법보다는 1.88% 높은 압축률을 나타냈다.

5. 결론 및 향후 연구 방향

본 논문에서는 full-text 문서 검색 시스템을 위한 비트벡터를 압축할 때 기존의 Bit Tree 방법에서 최적의 압축을 위한 block 크기를 결정하는 방법을 제안했다. Block의 크기는 전체 비트벡터에서 1의 값을 가지는 비트의 빈도수의 역수값에서 작거나 같은 가장 가까운 2의 승수 값으로 구해졌다.

그리고 비트벡터에서 1의 값을 가지는 비트의 위치를 분석하여 가변적인 기억공간을 사용함으로써 기존의 prefix omission을 이용한 압축 방법을 개선시켰다. 개선된 방법의 압축률은 Run Length에 비해 13.65, 기존의 Bit Tree의 압축 기법에 비해 1.88% 높게 나타났다. 또한 개선된 방법은 Run Length나 기존의 Bit Tree방법에 비해서 속도 차이는 거의 없었다.

비트벡터는 0과 1의 값을 가지므로 해당 term의 존재 유무만을 검색할 수 있다. 그러나 보다 효율적인 검색 시스템을 위해서는 빈도수나 휴리스틱 등에 의한 가중치 값까지도 정보로 가지는 구조에 대한 연구가 필요하다.

6. 참고문헌

[1] Y. Choueka, A.S. Fraenkel, S.T. Klein, E. Segal, *"Improved Hierarchical Bit-Vector Compression in Document Retrieval Systems"*, Organization of the 1986-ACM Conference on Research and Development in Information Retrieval, pp88-96, 1986.

[2] Alistair Moffat, Justin Zobel, *"Parameterised Compression for Sparse Bitmaps"*, In Proc. 15'th ACM-SIGIR Conference on Research and Development in Information Retrieval, pp274-285, June 1992.

[3] P. Elias, "Universal codeword sets and representations of the integers", IEEE Transactions on Information Theory, IT-21:194-203, March 1975.

[4] William B. Frakes, Ricardo Baeza-Yates, *Information Retrieval Data Structures & Algorithms*, Prentice Hall, 1992.

[5] 정영미, *정보검색론*, 성음사

[6] 김민정, 권혁철, "한국어 특성을 이용한 자동 색인 기법", 정보과학회 가을 학술발표 논문집, pp.1005-1008, 1992.