문자 해시와 이원 탐색 트리를 이용한 어절 빈도 계산 알고리즘의 성능 개선

박일남, 강승식
국민대학교 전자정보통신대학 컴퓨터공학부
e-mail : pin0156@naver.com, sskang@kookmin.ac.kr

Improvement of algorithm for calculating word count using character hash and binary search tree

Il-Nam Park, Seung-Shik Kang
School of Computer Science, College of EECS, Kookmin University

요 약

인터넷 검색 사이트는 사용자들이 검색한 단어들의 순위를 매기는 실시간 검색 순위 서비스를 제공하는데 검색되는 단어들의 순위를 매기기 위해서는 각 단어들의 분포도를 알 수 있는 어절 빈도 계산을 수행해야 한다. 어절 빈도는 BST(Binary Search Tree)를 수행하여 계산할 수 있는데, 사용자에 의하여 검색되는 단어들은 길이와 그 형태가 다양하여 빈도 계산시에 BST 의 깊이가 깊어져서 계산 시간이 오래 걸리게 된다. 본 논문에서는 문자 해시를 이용하여 깊이가 깊은 BST 의 탐색 속도를 개선하는 알고리즘을 제안하였다. 이 방법으로 빈도 계산 속도를 비교하였을 때 문자 해시의 범위에 의해 1KB 의 추가적인 기억공간의 사용하여 9.3%의 성능 개선 효과가 있었고, 해시 공간을 10KB 추가로 사용할 때는 24.3%, 236KB 일 때는 40.6%로의 효율로 BST 의 빈도 계산 속도를 향상시킬 수 있었다.

1. 서론

어절 빈도는 언어처리에 있어 각 단어들이 얼마나 분포하는지 알 수 있는 중요한 통계 정보이다. 어절 빈도 연산 수행은 자료들에 대하여 업데이트의 필요성이 있을 때만 수행하였으나 최근에는 많은 검색 포탈 사이트들이 실시간 검색 순위를 도입하여 많은 사용자들이 검색한 단어들을 실시간으로 순위를 정하여 보여주는 서비스를 제공하고 있다. 이러한 실시간 검색 순위를 위해서는 빈도 연산이 실시간으로 업데이트되고 사용자에게 보여준다는 것을 의미하며 웹에서의 사용자들이 검색한 단어들은 형태는 무한하며 매우 방대하다. 이러한 방대한 데이터들을 실시간으로 연산을 하므로 어절 빈도 연산의 속도를 개선할 필요성이 있다.

본 논문에서는 이진 탐색 트리(BST: Binary Search Tree)를 이용한 어절 빈도의 연산에서 탐색 속도를 개선하는 알고리즘을 제안한다. 논문의 구성은 2 절에서 BST 의 문제점을 설명하고, 3 절은 문자 해시를 이용하여 BST 의 탐색 속도를 줄이는 알고리즘을 기술하며, 4 절에서는 BST 와 제안한 방법과 비교 실험 결과를 기술한다.

2. BST 의 문제점

BST 는 현재 노드가 가지고 있는 값을 기준으로 입력된 값이 작으면 왼쪽 자식 노드로, 큰 값이면 오른쪽 자식 노드로 위치시키는 대표적인 트리 형식의 자료구조이다. 작은 값은 왼쪽에 큰 값은 오른쪽에 있으므로 자료의 삽입과 동시에 in-order 순회를 하게 되면 크기 순으로 정렬이 자동으로 되어있어 사전식 배열에 효율적인 알고리즘이다. BST 의 높이는 평균적으로 O(log*n*)이 될 수 있으며, 트리의 높이가 h 가 되는 BST 의 탐색 속도는 O(h)의 속도로 트리의 깊이가 깊을수록 탐색 속도가 떨어지는 단점을 가지고 있다 [1,2].

언어 처리에 있어 음절 빈도(character count)는 영문자 “A~Z”와 한글 기준으로 “가~힝”으로 그 개수가 정해져 있어 BST 를 수행하는데 그렇게 많은 시간이 들지 않는다. 하지만 어절 빈도(word count)는 개수가 한정되어 있는 음절 빈도와는 다르게 데이터들이 정해져 있지 않는 임의의 무한한 단어에 대하여 BST 를 수행하게 되므로 트리의 깊이가 무한대가 될 수 있으므로 탐색 시간이 비효율적이다.

3. 문자 해시와 BST 를 이용한 개선 방법

3.1 문자 해시와 BST 를 이용한 방법 제안

음절 빈도를 계산하는 방법 중 가장 빠른 방법은 해당 문자에 대하여 아스키 코드의 십진수 표기법으로 그 자체가 해시의 인덱스가 되는 방법이다[3]. 아스키 코드의 표현 범위는 확장 아스키 포함 256 개이므로 배열의 크기를 256 개 할당하고 다음과 같이 해

시함수로 표현할 수 있다.

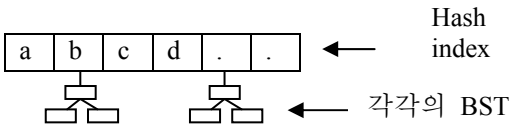
f[x]++; //x 는 아스키코드 문자

'a'가 입력으로 주어지면 해당 연산은 f['a']++; 로서 십진수로 표현하면 f[97]++; 이므로, 배열의 97 번지의 값을 1 증가시키는 결과를 나타낸다. 이러한 문자 해시는 해당하는 값이 들어왔을 때 배열로 맵핑시켜주는 연산이 따로 필요치 않으므로 시간 복잡도가 O(1)이 된다.

어절 빈도를 음절 빈도처럼 해시를 사용하면 속도를 빠르게 개선시킬 수 있으나 임의의 서로 다른 단어 100 개가 출현하면 해시는 그에 대응하는 100 개가 필요할 것이며, 무한대의 서로 다른 단어에 대하여 무한한 메모리를 사용하므로 결국 속도는 개선시킬 수 있으나 기억공간의 활용 측면에서는 효율적이지 못하다. 이러한 문제는 정해질 수 있는 범위에 한정하여 해시함수를 설계함으로써 해결할 수 있다.

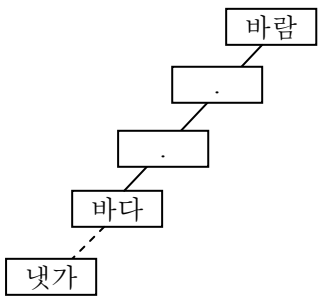
단어들의 개수는 무한하며 그 길이도 무한하다. 하지만 영, 한글 단어의 첫 글자만 봤을 때 글자의 종류와 길이가 "A~Z", "가~힝"으로 한정되며 범위의 기준이 뚜렷해진다. 단어의 첫 글자를 이용하여 문자 해시 수행 방법은 다음과 같다.

문자 해시를 통하여 해당 영역으로 분류된 단어들은 각자 해당되는 영역 안에서 BST 를 수행하게 한다. 그러면 알 수 없는 길이의 단어에 대하여 적어도 첫 글자에 대한 O(logn)탐색 속도를 O(1)로 줄일 수 있다.

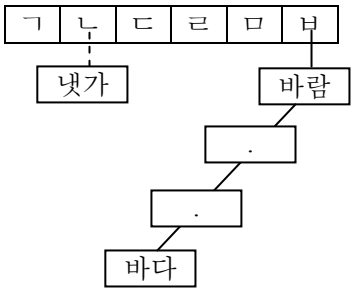


(그림 1) 문자 해시와 이진트리를 이용하는 방법

그림 2 처럼 구성되어 있는 BST 가 있다고 하자, "바람"과 "바다" 사이의 노드의 개수는 100 개라고 가정했을 때 "냇가" 삽입 연산에 있어 트리의 탐색 속도는 100 이 된다. 하지만 그림 3 처럼 단어의 첫 글자에 대하여 분류를 한 후 BST 연산을 하면 "냇가"의 삽입에 있어 탐색 횟수가 1 이 될 수 있음을 보여준다.



(그림 2) "냇가" 삽입 시 BST 구조



(그림 3) "냇가" 삽입 시 문자 해시와 BST 구조

3.2 문자 해시와 BST 를 이용한 개선 알고리즘

하나의 문자는 컴퓨터상의 표현방식으로 영문은 1byte 한글은 2byte 로서 그 표현 방법이 다르므로 해시 함수를 구현할 때 다음과 같이 각각 분류하여 설계하였다.

(1) 어절의 첫 글자가 1byte 문자의 경우 영문 또는 아스키 코드에 기반하는 범위이다. 이 문자들은 아스키 코드 10 진수 표기법으로 0~255 표현할 수 있다.

(2) 2byte 문자의 경우 한글완성형 코드에 기반하여 한글영역 상위비트(B0~C8) : 25 개 하위비트(A1~FE) : 94 개를 사용하여 총 2350 의 수로 표현할 수 있다.

(3) 2byte 문자 한글완성형 코드에 기반한 한글영역 이외의 문자인 경우 한글 범위 이전의 위치한 특수기호 부분(상위비트 A1~AC), 한글 범위 이후에 위치한 고어(상위비트 CA~FD), 한글완성형 코드가 아닌 2byte 문자를 처리하여야 한다.

```
Algorithm BST_add_CharHash()
{
char **words;       //입력된 단어들
int index;          //첫 글자에 따라 연산된 index

int *ascil[256];    //1 바이트 문자 해시 영역 index
int *euc_kr[25];    //2 바이트 한글 코드 해시 영역 index
int *pre_temp;      //2 바이트 한글 이전 해시 영역 index
int *post_temp;     //2 바이트 한글 이후, 완성형 한글코드
                    이외의 문자에 대한 해시 영역 index

for( i = 0; i < 입력 단어수; i++)
{
  char *p = words[i][0];        //삽입할 단어의 첫 글자

  if (*p & 0x80) {              //2 바이트 문자 영역
    if ((*p – 0xB0) < 0) {      //pre_temp 영역
     /* pre_temp index 영역에서의 BST 연산 */
     Insert_binary(words[i], pre_temp);
    } else if ((*p – 0xC8) > 0) {           //post_temp 영역
     /* post_temp index 영역에서의 BST 연산 */
     Insert_binary(words[i], post_temp);
    } else {        //한글 완성형 한글 영역
     index = (*p – 0xB0); //한글에 대한 index 계산
```

```
        /* euc_kr[index] 영역에서의 BST 연산 */
        Insert_binary(words[i], euc_kr[index]);
      }
    } else {            //1 바이트 문자 영역
      index = *p;

      /* ascil[index] 영역에서의 BST 연산 */
      Insert_binary(words[i], ascil[index]);
    }
  }
```

(그림 4) 문자 해시와 BST 를 이용한 어절 빈도 계산 알고리즘

그림 4 의 알고리즘에 따라 "가방"이라는 어절이 입력되었을 때 첫 글자 '가(B0A1)'는 한글 완성형 코드의 한글 2 바이트 문자로 상위비트 B0 을 이용하여 index = ('가'의 상위바이트 – 0xB0); 의 식으로 표현할 수 있으며 index 는 0 이라는 값을 나타내게 된다. 그러므로 '가'의 경우 euc_kr[0]이 가리키는 영역에서 BST 연산을 수행하게 된다.

4. 실험 및 비교 평가

4.1 실험 환경

<표 1> 실험 PC 사양

구분	성능
CPU	Intel® core™2 duo CPU E6750@2.66GHZ
RAM	3.50GB

<표 2> 실험 말뭉치

File name	Total word (어절)	Unique word (어절)	File Size (MB)
한겨레	5,441,375	938,439	42
한국일보	11,214,427	1,196,293	102
세종	57,120,248	5,828,137	420

한겨레 말뭉치는 2002 년도 1 년치의 한겨레 기사를 모아놓은 데이터이며, 한국일보 말뭉치는 1998 ~ 1999 년의 한국일보 기사를 모아놓은 데이터이다. 세종 말뭉치는 21 세기 세종계획 과제에서 구축된 말뭉치로 영, 한글, 고어로 이루어져 있다[4].

4.2 실험 방법

해시에 사용되는 기억공간의 크기에 비례하여 연산 속도를 얼마나 개선시킬 수 있는지를 측정하고자 위에서 제안한 알고리즘에서 한글 영역부분에 대한 해시 함수를 어절의 첫 1 바이트, 2 바이트, 3 바이트로 해시함수의 범위를 구성하여 범위의 증가에 따른 탐색 속도를 측정하고 각각의 어절의 수가 다른 말뭉치에 대하여 기존의 one root 방법인 BST 를 기준으로 각각 효율성이 얼마나 좋은가를 측정 한다.

(1) A 방법 : 하나의 root 를 갖는 기존의 BST 방법이다.

(2) B 방법 : 한글완성형 코드에 기반하여 어절의 첫 1 바이트를 해시 함수로 이용하는 방법으로 한글영역 상위비트(B0~C8) : 25 개를 사용한다. 그림 4 알고리즘에서 제시한 방법이다. 사용된 기억 공간의 크기는 아스키코드영역 (256) + 한글영역(25) + 기타(2)로 283 개의 공간이 필요하며 int 형(4byte)로 공간을 표현하였으므로 283*4byte 로 A 방법보다 약 1kbyte 기억공간을 더 사용하게 된다.

(3) C 방법 : 한글완성형 코드에 기반하여 어절의 첫 2 바이트를 해시 함수로 이용한다. 상위비트 (B0~C8) : 25 개, 하위비트(A1~FE) : 94 개를 사용하여 총 2350 의 수로 표현할 수 있다. 기억 공간의 크기는 (2)에서 연산하였던 방법으로 하여 (256 + 2350 + 2) * 4byte, 약 10kbyte 기억공간이 필요하다. 해시 함수는 그림 4 의 한글영역에 대한 index 계산을 다음과 같이 작성하면 된다.

```
index = ((*p - 0xB0) * 94) + (*(p+1) - 0xA1);
```

(4) D 방법 : 한글완성형 코드에 기반하여 어절의 첫 3 바이트를 해시 함수로 이용하는 방법으로 (2)와 (3)에서 제시한 방법을 조합하여 25 * 94 * 25 로 총 58750 의 수로 표현할 수 있다. 기억 공간은 (256 + 58750 + 2) * 4byte 를 하여 약 236kbyte 의 기억공간이 필요하다. 해시 함수는 다음과 같다.

```
index = ((*p - 0xB0) * 94) + (*(p+1) - 0xA1) +
(*(p+2) - 0xB0);
```

4.3 실험 결과

표 3,4,5 은 해당하는 말뭉치에 대하여 A~D 방법에 대한 어절 빈도 연산 시의 탐색 속도를 측정한 결과이며 기존의 A 방식을 기준으로 탐색 시간이 얼마나 개선되었는지에 대한 상대 비교를 보여주고 있다.

<표 3> 한겨레 말뭉치에 대한 어절 빈도 연산 시간

방법	탐색 속도	상대 비교
A	3.57	-
B	3.24	10%
C	2.82	26%
D	2.26	58%

<표 4> 한국일보 말뭉치에 대한 어절 빈도 연산 시간

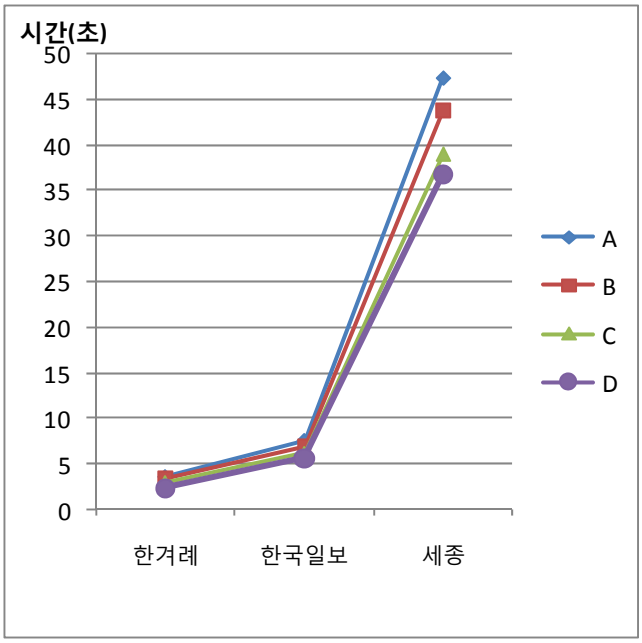
방법	탐색 속도	상대 비교
A	7.54	-
B	6.85	10%
C	6.02	25%
D	5.57	35%

<표 5> 세종 말뭉치에 대한 어절 빈도 연산 시간

방법	탐색 속도	상대 비교
A	47.43	-
B	43.89	8%
C	38.96	22%
D	36.64	29%

실험 결과를 보면 각각의 방법 별 말뭉치에 대하여 연산 속도가 A 방법보다 빠른 것을 볼 수 있다. 하지만 말뭉치 종류별로 보았을 때 B 방법은 10%, 10%, 9%, C 방법은 26%, 25%, 21%, D 방법은 58%, 35%, 29%의 효율로 탐색 속도의 효율이 평균에 미치지 못하는 것으로 나타났다. 이것은 어절의 첫 부분을 기준으로 영역을 구분한 뒤 BST 를 수행하여도 그 구분된 영역 내의 데이터들이 방대하게 많아지면 결국 해당 영역 안에서는 기존의 A 방법의 연산 속도와 비슷해진다는 것을 의미한다. 특히 세종 말뭉치는 다른 말뭉치들에 비해 탐색 속도가 빠르지 못하였는데, 이러한 이유는 세종 말뭉치는 영, 한글 단어 이외에 고어들이 많이 섞여있다. 본 논문의 알고리즘은 고어들에 대하여 하나의 영역으로만 분류하였기 때문에 첫 글자에 상관없이 모든 고어들은 한 개의 영역에서만 연산을 수행하였기에 이러한 실험 결과가 나온 것이다. 고어에 대한 영역도 한글 영역처럼 해시 범위를 늘린다면 이를 해결할 수 있을 것이다.

그림 5 는 실험 결과에 대하여 평균 탐색 속도를 기입하여 작성한 그래프이다. 그래프의 수직 축은 시간(초)을 나타내며 수평 축은 어절의 개수가 작은 순부터 큰 순으로 나열한 말뭉치 종류이다. 그래프를 보면 해시의 범위가 클수록 어절 빈도 연산이 빠르다는 것을 알 수 있다.



(그림 5) 연산 속도 비교 차트

표 6 은 해시 범위를 위하여 사용된 기억 공간의 크기에 비례하여 얻어지는 평균 탐색 속도 효율성을 정리한 표이다.

<표 6> 해시 범위를 위한 추가 기억 공간에 비례하는 탐색 속도의 효율성

방법	사용된 기억 공간의 크기 (추가 root 의 수)	효율성
A	-	-
B	1KB (283)	9.3%
C	10KB (2608)	24.3%
D	236KB (59008)	40.6%

A 방법을 기준으로 비교하였을 시 해시함수를 구성하기 위한 기억 공간의 크기를 1KB 추가적으로 사용한 B 방법은 10%의 효율을, 10KB 를 추가적으로 사용한 C 방법은 24%의 효율을, 236KB 를 사용한 D 방법은 40%의 효율을 보였으며, 어절의 첫 글자에 대한 문자 해시 함수의 범위를 늘릴수록 BST 의 탐색 속도에 대하여 시간을 단축시킬 수 있음을 알 수 있다.

5. 결론

본 논문에서는 형태와 길이가 정해져 있지 않은 수많은 단어들에 대하여 BST 연산시 탐색 시간을 줄이고자 연산 수행 전에 영, 한글 단어의 첫 글자를 통하여 정해질 수 있는 범위로 한정하고 문자 해시를 통하여 영역별로 분류한 후 BST 를 수행하므로서 탐색 시간을 줄이는 알고리즘을 제안하였다. 한글 영역 부분의 해시함수 범위를 증가시켜 실험해 본 결과 해시의 범위에 따른 기억 공간의 크기를 1KB, 10KB, 236KB 를 사용하였을 때 어절 빈도 계산 속도는 각각 평균 9.3%, 24.3%, 40.6%로 속도가 향상되었다.

참고문헌

[1] 김숙영, “확률적 이진 검색 트리 성능 추정”, 컴퓨터産業敎育學會論文誌 ,v.2 no.2. pp.203-210, 2001

[2] 이석호 역, C++ 자료구조론 2nd edition. 인피니티북스, p.255-263, 2007.

[3] 이석호 역, C++ 자료구조론 2nd edition. 인피니티북스, p.403-408, 2007.

[4] 문화관광부, 21 세기 세종계획 국어 기초자료 구축, 2000.