韓國 컴퓨터情報學會 論文誌
第12卷 第4號, 2007. 9.

2007-12-4-2-6

정규 허프만 트리를 이용한 허프만 코드의 효율적인 디코딩

박 상 호*

Efficient Huffman Decoding using Canonical Huffman Tree

Sangho Park*

요 약

본 논문에서는 정규 허프만 트리의 성질을 이용하여 허프만 코드를 효율적으로 디코딩 하는 방법을 제안하고 그 특성을 살펴보았다. 허프만 코드를 정규 허프만 트리로 변환한 후 각 레벨의 단말 노드 값들에 대한 규칙을 이용하여 최소한의 정보로 허프만 트리를 표현하였다. 제안한 방법은 기존의 방법과 동일한 수행 시간을 유지하며 트리정보를 위한 메모리를 줄일 수 있었다. 트리정보를 저장하기 위한 메모리의 크기는 $2h + 2k\log n$로서 기존의 방법들보다 적었으며 심벌로 변환하기 위한 검색의 수는 비슷하였다.

Abstract

We present an efficient decoding scheme for Huffman codes in which we use a properties of canonical prefix tree. After Huffman tree is converted to canonical Huffman tree, we represent Huffman tree with minimum information using rules associated with values of nodes in canonical tree. The proposed scheme can reduce memory to store Huffman tree information while maintains the same processing time. The memory size in order to represent tree information is $2h + 2k\log n$ which is less than those of previous methods. But the number of search is similar to previously proposed techniques.

▸ Keyword : Huffman code, Decoding algorithm

• 제1저자 : 박상호
• 접수일 : 2007. 8.4, 심사일 : 2007. 8.18, 심사완료일 : 2007. 8.28.
* 안동대학교 전자정보산업학부 부교수

Ⅰ. 서 론

데이터 압축기술의 발달은 멀티미디어 파일의 저장과 전송을 가능하게 하여 멀티미디어 시대를 가능하게 하였다. 허프만 코딩[1]은 가장 널리 사용하며 매우 효과적인 압축 방식으로 텍스트, 이미지, 비디오 데이터의 손실(lossy) 압축 파일들의 무손실(lossless) 압축방법으로 널리 사용되고 있다 [2]. 허프만 코딩은 임의의 발생확률을 갖는 심벌 분포에서 최소의 엔트로피를 갖는 최적의 코드북(codebook)을 찾는 문제이다. 코드북은 심벌과 코드간의 일대일 매핑 함수라고 볼 수 있다. 허프만 코딩 기법이 제안된 이 후 허프만 코드의 인코딩과 디코딩에 대해 많은 연구결과들이 발표되었다[3-5, 16]. 이러한 방법들은 어떠한 전치부호(prefix code)에도 적용 가능하다.

멀티미디어 파일의 이용은 원 데이터를 압축 후 저장을 하거나 전송을 하는데 사용자가 매번 압축된 파일을 디코딩하여 사용한다. 따라서 인코딩(압축)하는 회수 보다 디코딩하는 횟수가 월등히 많아진다. 인코딩은 멀티미디어 파일을 판매하는 회사가 하게 되므로 압축방식 및 압축장비가 고가이며 압축시간이 어느 정도 시간을 요하여도 문제가 없다. 그러나 디코딩은 대부분 엔드유저가 사용하므로 디코딩 방식이 간단하며 많은 시간을 요하지 않는 것이 좋다. 허프만 코드에서도 인코딩 방식보다 효율적인 디코딩 방식을 찾아내는 것이 더 중요하다. 디코딩 방식은 인코딩 방식에 종속적인 경우도 있고 독립적인 경우도 있다.

본 논문에서는 허프만 코드의 디코딩 시간은 기존의 방법[12, 13]과 동일한 수준을 유지하며 허프만 트리의 정보를 전송하거나 저장하기위한 메모리를 줄일 수 있는 방법을 제안한다. [12]의 자료구조를 정규 허프만 트리(canonical Huffman tree) [14-15]에 적용하여 효율적인 디코딩 방법을 제안한다. 제안하는 기법은 [12]와 [13]보다 허프만 트리에 대한 정보의 양이 적어 압축파일의 크기를 줄일 수 있고 정보의 양이 적으므로 테이블에서 정보를 검색하는 시간이 짧아져 허프만 코드의 디코딩 시간을 비슷하게 유지할 수 있다.

Ⅱ. 기존의 방법들에 대한 고찰

2.1 효율적인 허프만 디코딩을 위한 기존의 방법들

Sieminski [8]은 압축된 파일의 개별 비트 단위로 디코딩하지 않고 몇 개의 비트들을 묶어 처리함으로서 디코딩 시간을 단축하였다. 그러나 그 방법은 디코딩 시간을 줄일 수 있으나 사전(dictionary)의 입력 데이터를 위하여 허프만 코드의 직접적인 디코딩 방식보다 더 많은 메모리가 필요하다. Tanaka [7]은 디코딩의 속도를 높이기 위하여 새로운 자료구조와 알고리즘으로 유한상태 오토마타(finite-state automata) 개념을 도입하였다. Hirschberg [9]는 허프만 트리(Huffman tree)의 전체 정보를 테이블에 저장하지 않고 일부 만 저장하는 기법을 사용하여 메모리를 줄이는 디코딩 방식을 제안하였다. 이 방식은 디코딩 시간도 동시에 줄일 수 있었다.

Hashemain [10]은 허프만 트리의 검색시간을 줄이고 메모리 사용을 줄이는 방법을 제안하였다. 그는 허프만 트리의 레벨이 크면 트리정보를 저장하기 위한 메모리가 크므로 메모리의 사용을 줄이기 위하여 클러스터링 알고리즘을 사용하였다. 트리의 깊이가 긴 허프만 트리들은 클러스트링을 통하여 트리의 깊이가 줄어들어 허프만 트리를 검색하는 시간을 줄일 수 있었다. 그러나 허프만 트리의 최적 클러스터링 방법을 찾는 것은 매우 어려운 일이며, 이는 공개된 문제(open problem)이다.

Chung [11]은 심벌의 개수가 n일 때 메모리 크기를 $2n-3$으로 허프만 트리를 표현할 수 있는 자료구조를 제안하였다. Chen [12]는 [11]의 방법을 개선하여 메모리의 사용을 더욱 줄였으나 각 심벌의 압축된 데이터의 비트 수를 알아야 한다. 그러나 일반적으로 저장하거나 전송하는 압축 파일에는 각 심벌들의 압축 데이터의 비트 수가 포함되지 않는다. [12]의 제안은 디코딩을 위한 자료구조의 제안으로 볼 수 있다. Chowdhury [13]은 [12]의 자료구조를 사용하여 압축된 데이터의 비트 수를 알 수 없는 실제 압축파일에 대한 디코딩 방법을 제안하였다.

2.2 허프만 트리의 메모리를 줄이기 위한 자료구조

T 를 n 개의 심벌로 이루어진 허프만 트리라고 하고, T의 단말들(terminals or leaves)인 심벌들을 왼쪽에서 오른쪽으로 $s_0, s_1, \cdots, s_{n-1}$ 이라 하자. T의 노드(node)의 레벨(level) l 은 트리의 근(root)는 레벨 0라 하고, 그 아래의 노드는 레벨 1, 또 그 아래의 노드는 레벨 2와 같이 위의 노드보다 아래의 노드가 레벨이 크다. 가장 큰 레벨을 허프만 트리의 높이(height) h 라 한다. 레벨 l에 있는 노

드의 무게(weight) w 는 2^{h-l} 이다. w_i를 s_i의 무게라 하고 $count$를 다음과 같이 정의하자. i = 1, 2, ⋯ , $n-1$에 대하여 $conut_0 = w_0$, $count_i = count_{i-1} + w_i$. 허프만 트리를 설명하기 위하여 [12]에서 사용한 허프만 트리를 그림 1에 나타내었고 각 노드 s_0, s_1, ⋯ , s_{n-1}의 무게 w_i 와 $count_i$ 를 표 1에 나타내었다.

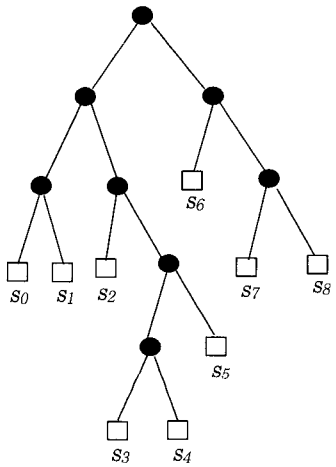

그림 1. 허프만 트리의 예
Fig. 1 An example of a Huffman tree

표 1. 노드의 w 값과 count 값
Table. 1 Values of w and count of nodes

s_i	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8
w_i	4	4	4	1	1	2	8	4	4
$count_i$	4	8	12	13	14	16	24	28	32

[12]에서 제안한 허프만 코드 디코딩 알고리즘은 아래와 같다.

Algorithm A
Input : The value of s_i , w_i and $count_i$, i = 0, 1, ⋯ , n-1, of a Huffman tree T with height h and a binary codeword c.
Output : The corresponding symbol s_k of c.
Method :
Step 1. Compute $t = (c + 1) \times 2^{h-d}$, where d is the number of binary digits in c.
Step 2. Search t from array count, if t is not in the array count, then c is not a codeword of T; otherwise assume that $count_k = t$.
Step 3. If $w_k \neq 2^{h-d}$, then c is not a codeword of T; otherwise s_k is the corresponding symbol of codeword c.
End of Algorithm A

[12]에서 제안한 알고리즘은 두 가지 단점이 있는데 심벌들을 압축하여 허프만 코드로 만들었을 때 각 심벌들에 해당하는 허프만 코드의 비트 수를 수신 측에 알려 주어야 하며 허프만 코드를 심벌로 매칭 할 때 w_i 와 $count_i$ 를 계산하여 표 1과 같은 테이블에 그 값이 있는지 비교하여야 한다. 허프만 코드의 비트수를 수신 측에 전송하는 것은 심벌의 수가 많을 경우 압축된 파일의 크기를 증가시키는 요인이 되며 허프만 코드를 심벌로 매칭 할 때 w_i 와 $count_i$ 를 계산하고 테이블의 값과 검색함으로서 디코딩 시간이 증가하게 되는 요인이 된다.

2.3 비트 수에 대한 정보가 주어지지 않는 경우의 디코딩 방법

[13]에서는 [12]에서 제안한 알고리즘을 이용하여 개선된 디코딩 알고리즘을 제안하였다. [13]에서는 전송하는 허프만 코드의 오버헤드를 줄이기 위하여 허프만 트리에 대한 축약된 정보만 전송하는 방법을 제안하였다. 허프만 트리 T의 노드 중에서 단말 노드(leaf node)에 해당하는 노드들을 제거한 허프만 트리 T'에서 단말 노드들에 대한 정보를 보내면 수신 측에서 T'을 이용하여 완전한 허프만 트리 T를 구성할 수 있다. [13]에서는 [12]에서 w_i 와 $count_i$ 를 계산하고 테이블의 값과 검색하여야 하는 과정을 $count_i$ 값을 계산한 후 테이블에서 $count_i$ 에 해당하는 심벌을 찾기 위하여 이진검색을 이용하였다. [13]에서 제안한 알고리즘은 아래와 같다.

Algorithm B
Input : The values f_i , i = 0, 1, ⋯ , n-1, of a Huffman tree T with height h and a bit stream b_j, j = 1, ⋯ , N
Output : The text symbols c_k , k = 1, ⋯ , M, corresponding to the input bit stream
// j is the pointer to the last bit already

decodable, k is the pointer to the last decoded symbol, *index* is the location where the search for d+1 fails. Otherwise, it is the location at which d+1 has been found. q_i is a stream of i 1 bits. '#' is the concatenation operator. //

$j \leftarrow 0$, $k \leftarrow 0$
$f_{-1} \leftarrow 0$
while $j < N$ do
  if ($h < N - j$) then
    $d \leftarrow b[j+1:j+h]$
  else
    $d \leftarrow b[j+1:N]\#q_{h-N+j}$
  endif
  binsearch(f, d+1, *index*)
  $k \leftarrow k+1$
  $c_k \leftarrow f_{index}$
  $j \leftarrow j + h - \log_2 (f_{index} - f_{index-1})$
End do
End of Algorithm B

위에 나타낸 [13]의 알고리즘은 $count_i$ 값을 검색하는 구체적인 검색알고리즘을 명시하지 않은 [12]의 자료구조에서 이진검색 알고리즘을 사용하여 디코딩 하는 방법을 제시한 것이다. 이 알고리즘에서 코드의 길이가 h 보다 적으면 다음 심벌의 $h - l$ 비트를 채워 h 비트의 코드로 만든 후 심벌을 찾는다. 그러나 검색알고리즘에서 심벌을 찾기 위한 기준 값인 $count_i$는 항상 $h - l$ 비트를 1로 채워야 하는데 다음 심벌의 코드에서 가져온 $h - l$ 비트는 항상 1이 아니므로 처음 가정과 달리 코드의 비트수를 알고 있어야 검색이 가능하다.

III. 정규 허프만 트리를 이용한 허프만 코드의 효율적인 디코딩

3.1 정규 허프만 트리

본 절에서는 [12]의 자료구조를 정규 허프만 트리(canonical Huffman tree)[14-15]에 적용하여 효율적인 허프만 트리 디코딩 방법을 기술한다. 정규 허프만 트리는 트리 노드가 왼 쪽에서 오른 쪽으로 code의 값이 커지게 한 전치부호(prefix code) 트리이다. 여기서 코드의 길이는 코드를 나타내는데 필요한 비트의 수이다. 그림 1의 트리에서 단말 노드(terminal node or leaf node)들의 코드의 길이는 [3 3 3 5 5 4 2 3 3]이다. 단말 노드의 코드길이는 그 노드의 레벨과 같다. 그림 1의 트리를 정규 허프만 트리로 바꾸면 단말 노드의 코드길이는 [2 3 3 3 3 3 4 5 5]가 된다. 정규 트리 코드는 몇 가지 중요한 성질이 있다. 각 레벨의 단말 노드에 할당된 코드는 연속적인 이진수이다. 레벨 l 의 코드 c_l 은 이전 레벨 l -1 의 마지막 코드 d_{l-1}과 $c_l = 2(d_{l-1} + 1)$의 관계가 있다. 만약 레벨 중 어떤 레벨에는 단말 노드가 없는 경우를 살펴보면, 레벨 1에 단말 노드가 있고 레벨 2에는 단말 노드가 없으며 레벨 3에 단말 노드가 있다면 레벨 3의 첫 번째 단말 노드의 코드의 값은 $c_3 = 2(2(d_1+1))$이다. 그림 1의 허프만 트리의 정규 허프만 트리를 그림 2에 나타내었다.

정규 허프만 트리에서 각 단말 노드들의 코드 값이 작을수록 그 심벌의 발생확률이 높다. 노드 *A*는 레벨 2에 있으므로 코드는 '00'이며 2비트이다. 노드 *B, C, D, E, F*는 레벨 3에 있으므로 코드는 3 비트이며 할당된 코드는 각각 '010', '011', '100', '101', '110'이다. 노드 *G*는 레벨 4에 있으므로 코드는 '1110'이며 4 비트이다. 노드 *H*와 *I*는 레벨 5에 있으므로 코드의 길이는 5비트이고 할당된 코드는 각각 '11110', 과 '11111'이다.

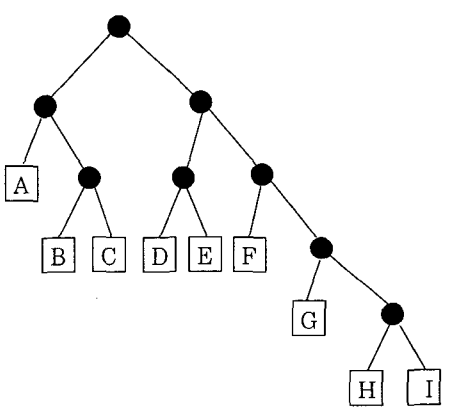

그림 2. 정규 허프만 트리의 예
Fig. 2 An example of a canonical Huffman tree

3.2 정규 허프만 트리를 이용한 디코딩

정규 허프만 트리를 이용하여 심벌을 압축하기 위해서 각 심벌들의 발생빈도에 따라 허프만 트리를 구성하고 발생

빈도가 제일 높은 심벌의 코드부터 트리의 왼쪽 노드에서 시작하여 발생빈도가 낮은 순서대로 심벌의 코드를 오른쪽으로 정렬하여 트리를 재구성한다. 정규 허프만 트리를 사용함으로서 단말 노드의 코드 값들은 왼쪽에서 오른쪽으로 오름차순으로 정렬되어있다. 수신 측에서는 전달된 비트 열에서 h 비트를 취하여 그 코드에 해당하는 심벌을 찾는다. 레벨 h 에 있는 코드들은 그 코드가 심벌을 찾기 위한 코드가 되며 레벨이 h 보다 높은 레벨인 레벨 1부터 레벨 h -1에 있는 노드들은 오른쪽의 h - l 비트는 다음 심벌의 비트들로 구성되어 있다. 여기서 h 는 트리의 높이이고 l 은 노드가 위치하고 있는 레벨번호이다.

허프만 트리의 레벨들은 코드 값의 범위가 정해지게 된다. 그림 2의 예에서 트리의 높이 h 는 5이며 심벌의 수 n = 9 이다. 노드 A는 레벨 2에 있으며 코드는 '00'이고 오른쪽 3 비트를 채울 수 있는 경우는 '000'부터 '111'이므로 5비트를 만들면 '00000'부터 '00111'이 되어 레벨 2의 노드 A의 코드 값의 범위는 0부터 7이다. 레벨 3에는 '010'부터 '110'의 5 개의 단말 노드가 있다. 따라서 레벨 3에는 가장 값이 작은 5 비트 코드 '01000'부터 가장 값이 큰 코드인 '11011'이 있어 레벨 3의 노드들의 값의 범위는 8부터 27이다. 같은 방법으로 레벨 4의 코드 값의 범위는 28부터 29이고 레벨 5의 코드 값의 범위는 30부터 31이다. 동일한 레벨의 노드들의 값은 오른쪽 h - l 비트들을 1로 채웠을 때 이웃한 노드들 간에는 2^{h-l} 차이가 난다. 예를 들면 그림 2에서 레벨 3에 5개의 단말 노드가 있다. 각 노드들의 코드의 오른쪽 2 자리를 1로 채우면 각 코드들의 값 f_{lk} 의 간격은 B는 11, C는 15, D는 19, E는 23, F는 27로 2^{5-3} = 4 이다. 정규 허프만 트리의 이러한 특성을 이용하여 허프만 트리의 각 레벨 중 단말 노드가 있는 레벨의 노드 중 레벨의 수 l, 코드의 최대값과 단말 노드의 수만 전송하면 수신 측에서 허프만 트리의 구성이 가능하다. 허프만 트리를 나타내기 위하여 $2h + 2k\log n$ 보다 적은 메모리가 요구된다. 여기서 k는 허프만 트리에서 노드가 있는 레벨의 수이다.

수신 측에서는 전송된 허프만 트리에 대한 정보를 이용하여 허프만 트리를 구성하고 정규 허프만 트리의 왼쪽 노드부터 그 노드의 최대값들로 테이블을 작성한다. 수신된 데이터 파일의 비트 열에서 h 비트 마다 코드에 해당하는 심벌을 찾고 만약 코드의 비트 수가 h 보다 작으면 디코딩한 h 비트의 코드에서 l+1 번째 비트부터 5비트를 구성하여 다음 심벌을 찾는다. 여기서 코드의 비트번호는 1부터 시작한다고 가정하였다. 제안된 알고리즘은 노드들의 블록을 먼저 검색한 다음 그 블록에 노드가 하나뿐이면 그 코드 값으로 심벌을 할당하고 블록 내에 2개 이상의 노드가 있으면 블록 내에서 코드에 해당하는 심벌을 찾는다. 본 논문에서 제안하는 디코딩 알고리즘은 아래와 같다.

Algorithm Canonical
Input : The values f_{lk} , l = 1, ⋯ , h, k is node number in the same level, of a Huffman tree T with height h and a bit stream b_j, j = 1, ⋯ , N
Output : The text symbols c_k, k = 1, ⋯ , M, corresponding to the input bit stream
// j is the pointer to the last bit already decodable, k is the pointer to the last decoded symbol, *index* is the location at which d has been found. 1_i is a stream of i 1 bits. '#' is the concatenation operator. //
$j \leftarrow 0$, $k \leftarrow 0$
$f_{-1} \leftarrow 0$
while $j < N$ do
  if ($h < N - j$) then
    $d \leftarrow b[j+1 : j+h]$
  else
    $d \leftarrow b[j+1 : N]$ # 1_{h-N+j}
  endif
  Search(f_{lk}, d, *level*)
  if number of nodes in the *level* is 1
    $index = \sum_{0}^{level-1}$ number of nodes
    $c_k \leftarrow f_{index}$
  else
    Search(f_{lk}, d, *index*)
    $c_k \leftarrow f_{index}$
  endif
  $k \leftarrow k+1$
  $j \leftarrow j + h - level$
End do
End of Algorithm Canonical

IV. 실험 및 분석

본 논문에서 제안한 알고리즘을 그림 2의 예를 이용하여 설명하기 위하여 스트링 ABDGH를 가정하자. 허프만 트리에 의하여 스트링은 '00010100111011110'의 비트 열로 코딩된다. 본 논문에서 제안한 방법으로 디코딩 하려면, 수신 측에서는 수신된 트리 정보와 비트 열을 이용하여 표 2와 같은 테이블을 만든다. 제안한 알고리즘과 기존의 디코딩 알고리즘간의 비교를 위하여 알고리즘 [12]와 [13]의 허프만 트리 테이블도 표 2에 나타내었다. [12]와 [13]은 동일한 허프만 트리 테이블을 사용하며 [12]에서는 허프만 코드의 비트 수를 추가로 제공하여야 하고 [13]은 제안한 알고리즘과 같이 허프만 코드의 길이에 대한 추가적인 정보가 필요 없다. 제안한 알고리즘과 기존의 알고리즘의 오버헤드의 크기를 비교하면 표 3와 같다. 여기서 h는 레벨의 크기, $k < n$ 인 임의의 수, n은 심벌의 수이다.

표 2. 허프만 트리 테이블
Table. 2 Huffman tree table

	node	A	B	C	D	E	F	G	H	I
제안한 방법	level	2	3	3	3	3	3	4	5	5
	f_{lk}	7	11	15	19	23	27	29	30	31
[12] [13]	w_i	8	4	4	4	4	4	2	1	1
	$count_i$	8	12	16	20	24	28	30	31	32

표 3. 트리정보를 나타내기 위하여 사용되는 메모리 크기 비교
Table. 3 Comparison of memory for overhead

	Memory space
Hashemian [10]	$O(n) \sim O(2^h)$
Chung [11]	$2n - 3$
Chen [12]	$\lceil 3n/2 \rceil + \lceil n/2 \log n \rceil + 1$
Chowdhury [13]	$O(n \log n) + kO(h)$
Our algorithm	$2h + 2k \log n$

비트 열을 심벌로 변환하기 위하여 제안한 디코딩 알고리즘은 트리의 높이 h인 5 비트 단위로 심벌을 검색한다. 제일 먼저 코드 '00010'을 구성하여 그 값을 계산하면 d = 2이므로 그 값이 7보다 적어 레벨 2임을 알 수 있고 실제 코드는 2 비트로서 '00'임을 알 수 있다. 레벨 2에는 단말 노드가 하나 밖에 없으므로 '00'은 A로 변환된다. 다음 심벌을 검색하기 위하여 수신 비트 열에서 세 번째 비트부터 5 비트로 코드 '01010'을 구성한다. d = 10이므로 레벨 3의 최대값 27보다 작으므로 심벌이 레벨 3에 있음을 알 수 있다. 레벨 3에는 5개의 단말 노드가 있음이 트리정보에 있으므로 레벨 3의 노드 중 8이상 11이하의 심벌 B를 찾아 B로 변환된다. 같은 방법으로 비트 열의 모든 코드들을 심벌로 변환할 수 있다.

디코딩 시간을 실험적으로 알아보기 위하여 그림 2의 허프만 트리를 이용하여 1000개의 심벌을 랜덤하게 발생하였다. 1000개의 심벌로 이루어진 입력 시퀀스를 허프만 트리를 이용하여 허프만 코드를 생성하였다. 본 논문에서는 허프만 코드의 디코딩 방식에 대해 논의하고 있으므로 생성한 동일한 허프만 코드를 사용하여 [13]과 제안한 알고리즘의 디코딩 수행시간을 비교하였다. 수행시간은 허프만 코드에서 메시지를 검색(search)하는데 필요한 검색 길이로 하였다. 허프만 코드의 비트 열에서 심벌을 찾기 위하여 [13]의 알고리즘과 제안한 알고리즘에서 허프만 트리의 노드를 검색하기 위하여 수행되는 검색 길이를 실험한 결과 [13]은 평균 2730회 제안한 알고리즘은 2770으로 비슷하였다. 허프만 트리의 노드를 찾는 방법에서 제안한 알고리즘과 [13]의 차이점은 [13]은 검색(search) 길이를 줄이기 위하여 이진 검색을 사용하도록 제한하였고 제안한 알고리즘은 검색방법을 제한하지 않는다. 알고리즘 구현 시 설계자가 데이터의 특성에 따라 검색 알고리즘을 선택하여 사용할 수 있다. 실험에서는 알고리즘 구현을 간략하게 하기위하여 선형 검색을 사용하였다.

V. 결 론

본 논문에서는 정규 허프만 트리의 특성을 이용하여 허프만 코드를 효율적으로 디코딩 하는 방법을 제안하고 그 특성을 살펴보았다. 허프만 코드를 정규 허프만 트리 모양으로 구성한 후 각 레벨의 단말 노드들 간의 값의 대한 규칙을 이용하여 최소의 정보를 이용하여 허프만 트리를 나타내었으며 트리정보를 저장하기 위한 메모리의 크기는 $2h + 2k \log n$로서 기존의 방법들보다 적었다. 그러나 심벌로 변환하기 위한 검색의 수는 비슷하였다.

참고문헌

[1] Huffman, D. A. (1952), "A method for the construction of minimum redundancy codes," *Proc. IRE*, Vol. 40, No. 9, Sept., pp. 1098-1101.

[2] Jain, A. K. (1981), "Image data compression: A review," *Proc. IEEE*, Vol. 69, March, pp. 349-389.

[3] Larmore L. L. and D. S. Hirschberg (1990), A fast algorithm for optimal length-limited Huffman codes," *Journal of the ACM*, Vol. 37, July, pp. 464-473.

[4] Knuth, D. E. (1983), "Dynamic Huffman coding," *Journal of algorithms*, Vol. 6, pp. 163-180.

[5] Vitter, J. S. (1987), "Design and analysis of dynamic Huffman codes," *Journal of the ACM*, Vol. 34, October, pp. 823-843.

[6] Cheung, G., S. McCanne, and C. Papadimitriou (1999), "Software synthesis of variable-length code decoder using a mixture of programmed logic and table lookups," In *Proceedings of the Data Compression Conference* (M. C. James A. Storer, Ed.), pp. 121-139, IEEE Computer Society, Los Alanitus, CA.

[7] Tanaka, H. (1987), "Data structure of Huffman codes and its application to efficient coding and decoding," *IEEE Trans., Information Theory*, Vol. 46, January, pp. 154-156.

[8] Sieminski, A. (1988), "Fast decoding of the Huffman codes," *Information Processing letters 26*, No. 5, May, pp. 237-241.

[9] Hirschberg, D. S. and D. A. Lelewer (1990), "Efficient decoding of Prefix codes," *Communications of the ACM*, Vol. 33, No. 4, pp. 449-459.

[10] Hashemian, R. (1995), " Memory efficient and high-speed search Huffman coding," *IEEE Trans., Communications*, Vol. 43, No. 10, October, pp. 2576-2561.

[11] Chung, K. L. (1997), "Efficient Huffman decoding," *Information Processing Letters* 61, pp. 97-99.

[12] Chen, H.-C., Y.-L. Wang, and Y.-F. Lan (1999), "A Memory-efficient and fast Huffman decoding algorithm," *Information Processing Letters* 69, pp. 119-122.

[13] Chowdhury, R. A., M. Kaykobad, and I. King (2002), "An efficient decoding technique for Huffman codes," *Information Processing Letters* 81, pp. 305-308.

[14] Schwartz, E. S. and B. Kallick (1964), "Generating a canonical prefix encoding," *Communications of the ACM*, Vol. 7, No. 3, March, pp. 166-169.

[15] Connell, J. B. (1973), "A Huffman-Shannon-Fano code," *Proceedings of the IEEE*, Vol. 61, No. 7, July, pp. 1046-1047.

[16] Kitakami, M. and Nakamura, S. (2005), "Burst error recovery for Huffman coding," *IEICE Trans. Inf. & Syst.*, Vol. E88-D, No. 9, Sept., pp. 2197-2200.

저 자 소 개

박 상호

경북대학교 전자공학과, 공학사
영남대학교 전자공학과, 공학석사
Syracuse University, MS
State University of New York at Buffalo, Ph. D.
현재: 안동대학교 전자정보산업학부 부교수