Chapter 9. Greedy Algorithms: Huffman codes

Introduction

Consider the problem of coding (in binary) a message consisting of a string of characters. We can use either fixed-length encoding (same length of code for each character in the alphabet) or variable length encoding. Two fixed-length encoding schemes are widely used now – ASCII and Unicode. ASCII codes are 8-bit long, and in Unicode each character is encoded by 16 bits. Unicode allows for encoding various alphabets, while ASCII code can encode only 256 characters. In general both ASCII and Unicode are not efficient in terms of memory. For example, Unicode extends all ASCII codes by 8 zeros in front, thus wasting a lot of memory.

Another source of inefficiency is the fact that less frequent characters are encoded with same length as more frequent characters. For example, consider a string of 10 characters. In Unicode they can be represented with 20 bytes. Let’s assume that there are only 3 unique characters in the string: ‘A’ occurs 6 times, ‘B’ occurs 3 times, and ‘C’ occurs only once. If we could use 14 bits for ‘A’, 16 bits for ‘B’ and 20 bits for ‘C’, the total bit length would be 6*14 + 3*16 + 1*20 = 84 + 48 + 18 = 152 and this is 19 bytes instead of 20 bytes. This observation is in the heart of the well known and widely used Huffman codes used for data compression.

Overview of Huffman encoding

Huffman coding is a statistical technique which attempts to reduce the amount of bits required to represent a string of symbols. It is based on the insight that if we encode frequent characters in a text with shorter binary strings we will save memory. Thus the codes for each symbol in the alphabet vary depending on the frequency of the characters in the text to be encoded.

Building a Huffman Tree

The Huffman code for an alphabet (set of symbols) may be generated by constructing a binary tree with nodes containing the symbols to be encoded and their occurrence frequency. The tree may be constructed as follows:

1. Compute the frequencies of each character in the alphabet
2. Build a tree forest with one-node trees, where each node corresponds to a character and contains the frequency of the character in the text to be encoded
3. Select two parentless nodes with the lowest frequency
4. Create a new node which is the parent of the two lowest frequency nodes.
5. Label the left link with 0 and the right link with 1
6. Assign the new node a frequency equal to the sum of its children's frequencies.
7. Repeat Steps 3 through 6 until there is only one parentless node left.
Here is a simple example. Text: “here is a simple example”

Letters with their frequency:

<table>
<thead>
<tr>
<th>Letter</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>_ (space)</td>
<td>4</td>
</tr>
<tr>
<td>M</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>P</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>5</td>
</tr>
<tr>
<td>R</td>
<td>1</td>
</tr>
<tr>
<td>H</td>
<td>1</td>
</tr>
<tr>
<td>S</td>
<td>2</td>
</tr>
<tr>
<td>I</td>
<td>2</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>L</td>
<td>2</td>
</tr>
</tbody>
</table>

In the Figure below, the red links are labeled with “0” and the blue links are labeled with “1”. The colored nodes are the initial one-node trees.

The code for each symbol may be obtained by tracing a path from the root of the tree to that symbol.
Symbols with higher frequency will be included later in the tree and therefore will be closer to the root – this means shorter code.

**Implementation**

There are various implementations of the algorithm. One implementation is given below:

Let N be the number of the characters in the alphabet

Data structures need:

**Array frequencies[0..2N]** : represents node frequencies.

- The first N elements represent the frequencies of the alphabet characters.
  - if frequencies[k] > 0, 0 ≤ k ≤ N-1, then frequencies[k] is a terminal node representing the k-th character

**Array parents[0..2N]** : represents the parents of each node in array frequencies.

- The parent of node k with frequency frequencies[k] is given by abs(parents[k]).
  - If parents[k] > 0, node k is linked to the left of its parent, otherwise – to the right.

**Priority queue** with elements (k, frequencies[k]), where the priority is represented by frequencies[k].

1. For each character at position k in the alphabet compute frequencies[k] and insert in a priority queue if frequencies[k] > 0
2. m ← N
3. while PQueue not empty do
   a. deleteMin from PQueue (node1, frequency1)
   b. if PQueue empty break
   c. else deleteMin from PQueue (node2, frequency2)
   d. frequencies[m] ← frequency1 + frequency2 // new node
   e. parents[node1] ← m // left link
   f. parents[node2] ← -m // right link
   g. insert in PQueue (m, frequency1 + frequency2)
   h. m ← m + 1
4. end // tree is built with root = node1
To encode a character
Start with the leaf corresponding to that character and follow the path to the root
The labels on the links in the path will give the reversed code.

code ← “”
    while k not equal to the root do
        if parent[k] > 0 code = “0” + code
        else code = “1” + code
        k ← parents[k]
        if k < 0 k ← -k

To restore the encoded text
Start with the root and follow the path that matches the bits in the encoded text, i.e. go left if ‘0’, go right if ‘1’.
Output the character found in the leaf at the end of the path.
Repeat for the remaining bits in the encoded text

p ← 0
while p ≤ encodedTextSize
    k ← root
    while k > N-1 do
        if encodedText[p] is 0 find parents[m] so that parents[m] = k
        if encodedText[p] is 1 find parents[m] so that parents[m] = -k
        k ← m, p ← p+1
    output alphabet[k]

After the inner loop k will point to the encoded character in the alphabet.

Analysis of the algorithm to create the Huffman tree for an alphabet with length N:
O(NlogN)

The insert and delete operations each take log(N) time, and they are repeated at most 2N times (Why 2N? Answer: The leaves are N, therefore the nodes are less than 2N)

Therefore the run time is O(2NlogN) = O(NlogN)
**Discussion and Summary**

- Huffman trees give prefix-free codes. Prefix-free codes have the property that no code is a prefix of any other code. This allows to avoid using a delimiter to separate the codes of the individual characters.

- Huffman trees are full trees – i.e. each node except the leaves has two children.

- The length of the encoded message is equal to the weighted external path length of the Huffman frequency tree.

**Definition** Let $T$ be a tree with weights $w_1,...,w_n$ at its leaf nodes. The *weighted leaf path length* $L(T)$ of $T$ is defined as the sum

$$L(T) = \sum_{i \in \text{leaf}(T)} l_i w_i$$

where $\text{leaf}(T)$ is the set of all leaves of $T$, and $l_i$ is the **path length** - the length of the path from the root to node $i$.

In fact, Huffman codes solve the more general problem: Given a set of weighted leaves, construct a tree with the minimum weighted path length.

- **Optimality:** No tree with the same frequencies in external nodes has lower weighted external path length than the Huffman tree.
  
  This property can be proved by induction (proof is omitted here).

- The tree must be saved and sent along with the message in order to decode it. Thus Huffman code is effective for long files, or in situations where the coding tree can be pre-computed and used for a large number of messages. In general, the need to transmit the coding tree as well as the message reduces the effectiveness of the method a little. It can be impractical to preprocess a message to get the exact frequencies of the symbols before any of the message is transmitted. There is a variant however, called *adaptive* Huffman coding, in which the frequencies are assumed initially to be all the same, and then adjusted in the light of the message being coded to reflect the actual frequencies.

- For truly random files the code is not effective, since each character will have approximately same frequency and we’ll get a balanced coding tree with almost equal number of bits per letter.

- Widely used coding schemes such as zip (or gzip or pkzip) are based on the Huffman encoding.

A copy of one David Huffman's original publications about his algorithm may be found at [http://compression.graphicon.ru/download/articles/huff/huffman_1952_minimum-redundancy-codes.pdf](http://compression.graphicon.ru/download/articles/huff/huffman_1952_minimum-redundancy-codes.pdf)