Huffman Encoding

Given a text (or a block of data) made up of symbols (letters) from a set (alphabet) A, we wish to transmit this using 0's and 1's. This requires a one-to-one map $e : A \to \{0,1\}^+$, where the latter consists of non-empty sequences of 0's and 1's. Such a map is referred to as an *encoding* of the symbols in A.

At the receiving end, the *reverse* map is applied in order to recover the original text. Note that this appears to require indicators that show where the sequence has to be "cut" in order to obtain the elements in the image of the encoding. This can be avoided in two ways:

Fixed-length encoding: Ensure that $e(A)$ is a subset of $\{0,1\}^\ell$ for some fixed length ℓ.

Prefix-free encoding: Ensure that for every $a \in A$, the sequence $e(a)$ is not a prefix (initial sequence) of $e(b)$ for any b different from a.

To clarify the idea of prefix-free encoding, consider the map $e : \{a,b,c,d\} \to \{0,1\}^+$ defined by:

$$a \mapsto 10; b \mapsto 0; c \mapsto 110; e \mapsto 111$$

This is prefix-free. Moreover, any string of 0's and 1's has an element of $e(A)$ as a prefix.

In each text, the elements of A occur with some frequency. More generally, in all possible texts, one may have some idea of the frequency with which each element of A occurs. For example, it is well known that in English language texts, the letter `e` occurs more frequently than the letter `z`. Let $f : A \to (0,1)$ be the map specifying this (estimated) frequency.

One way to reduce the cost of transmission is choose e so as to minimize $\sum_{a \in A} f(a)L(e(a))$, where $L(s)$ is the length of a string s of 0's and 1's. An optimal such e is called a Huffman encoding. We will discuss the method proposed by D. A. Huffman to find such an encoding given the function f as above.

It seems reasonably clear that the more frequently a symbol $a \in A$ is, the smaller we wish $L(e(a))$. Thus, if we make a tree where the longer sequences are at greater depth, then the deepest sequences should correspond to the symbols with the lowest frequencies. This is the idea behind the following Python program.

The data `cf` is a list of pairs (f, c) where f is the frequency of the character c.

The program begins by making a min-heap `que` out of this data so that the elements with lowest frequency are picked earlier. (The program makes use of the `push` and `pop` operations for the min-heap `que`.) The data associated with each node is a pair consisting of a frequency f and a dictionary that maps the characters "below" this node to strings of 0's and 1's.

In each iteration, we pick the two lowest frequencies and combine then using 0 as a pre-"label" for the first and 1 as a pre-"label" for the second. (It could also be done the other way around!).

At the end, we will have obtained a labelling for every character, and this labelling is our required encoding.

```
def hcode(cf):
    n = len(cf)
    que = [(a[0],{a[1]:''}) for a in cf]
    heapify(que)
    for i in range(1,n):
        lval, lc = pop(que)
        rval, rc = pop(que)
        val = lval+rval
        c = {}
        for k in lc.keys():
           c[k]='0'+lc[k]
        for k in rc.keys():
           c[k]='1'+rc[k]
        z = (val, c)
        push(que,z)
    ans = pop(que)[1]
    return ans
```

Given the input (where the frequency has been replaced by a “count” so as to use integer arithmetic):

```
cf = [(17, 'a'), (20, 'b'), (19, 'c'), (13, 'd'),\
     (19, 'e'), (28, 'f'), (20, 'g'), (30, 'h')]
```

This program produces the Huffman code:

```
{'h': '00', 'e': '010', 'c': '011', 'g': '100',\
    'b': '101', 'f': '110', 'd': '1110', 'a': '1111'}
```

Note that the code is not unique. For example, if A has 2^k elements and each character has the same frequency $1/2^k$, then one could encode by any bijection between A and $\{0, 1\}^k$.

Optimality

The basic ideas behind the proof that the above algorithm gives an optimal solution are as follows.

Huffman Tree. The Huffman algorithm builds a tree which proceeds by repeated “join” operations as described below.

To begin with the variable que consists of nodes corresponding to elements of A paired with their frequencies.

1. We pick x and y as distinct elements of que.

2. We create a parent node z for x and y with *assigned* frequency equal to the sum of the frequencies of x and y.

As a result of this join x and y are no longer parent-less, so are out of `que`. The new node z is an element in `que`.

Repeating this join operation $n-1$ times produces a (binary) tree T with elements of A as leaf nodes. The length of the code word for a leaf $a \in A$ is the depth $d_T(a)$ of this leaf.

One easily checks that the cost $c(T) = \sum_{c \in A} f(c) d_T(c)$ of such a tree is the sum of the frequencies of each parent node created in a join operation.

Induction. Given A as above, let x and y be distinct elements of A. We change A into A_1 by removing x and y and inserting a new symbol z such that $f(z) = f(x) + f(y)$ (the actual symbol z does not matter only its frequency does!). By induction on the length we can assume that the above algorithm produces an optimal T_1 for A_1. From the above description, if T is the Huffman Tree produced by the algorithm for A, then we have the equation

$$c(T) = c(T') + f(x) + f(y)$$

Thus, the "greedy approach" is to ensure that $f(x)+f(y)$ is the smallest amongst all possible choices of x and y in A. This is what the above program does.

Swap. Finally, we must show that if some other pair a and b of distinct leaf nodes is joined instead of the minimal pair x and y of leaf nodes, then the resulting tree T_2 will have a higher cost. Note that $d_{T_2}(a) \leq d_{T_2}(x)$ and $d_{T_2}(b) \leq d_{T_2}(y)$.

We assume without loss of generality that $f(x) \leq f(y)$ and $f(a) \leq f(b)$. Next, consider the tree T_1 obtained from T_2 by swapping x and a, and the tree T obtained from T_1 by further swapping y and b. We claim that $c(T_2) \geq c(T_1) \geq c(T)$.

If $f(x) = f(b)$, then, since x is the node with the least frequency and $f(a) \leq f(b)$, we have $f(a) = f(x)$. Since a and b are distinct, either $x \neq a$ or $x \neq b$. In either case, since y is the node distinct from x that has the lowest frequency amongst such nodes, we must have $f(y) = f(b)$ or $f(y) = f(b)$. Thus, if $f(x) = f(b)$, then all four left nodes have the same frequencies; hence, $c(T) = c(T_1) = c(T_2)$.

If $f(x) \neq f(b)$, then $x \neq b$. We calculate

$$\begin{aligned} c(T_2) - c(T_1) &= f(x)d_{T_2}(x) + f(a)d_{T_2}(a) - f(x)d_{T_1}(x) - f(a)d_{T_1}(a) \\ &= f(x)d_{T_2}(x) + f(a)d_{T_2}(a) - f(x)d_{T_2}(a) - f(a)d_{T_2}(x) \\ &= (f(x) - f(a))(d_{T_2}(a) - d_{T_2}(x)) \geq 0 \end{aligned}$$

Thus, $c(T_2) \geq c(T_1)$. Similar arguments can be used to show $c(T_1) \geq c(T)$.

This completes a sketch of the proof of correctness.