Notes 16 for CS 170

1 Data Compression via Huffman Coding

Huffman codes are used for data compression. The motivations for data compression are obvious: reducing time to transmit large files, and reducing the space required to store them on disk or tape.

Suppose that you have a file of 100K characters. To keep the example simple, suppose that each character is one of the 8 letters from a through h. Since we have just 8 characters, we need just 3 bits to represent a character, so the file requires 300K bits to store. Can we do better?

Suppose that we have more information about the file: the *frequency* which which each character appears. The idea is that we will use a *variable length code* instead of a *fixed length code* (3 bits for each character), with fewer bits to store the common characters, and more bits to store the rare characters. At one obvious extreme, if only 2 characters actually appeared in the file, we could represent each one with just one bit, and reduce the storage from 300K bits to 100K bits (plus a short header explaining the encoding). It turns out that all characters can appear, but that as long as each one does not appear nearly equally often (100K/8 times in our case), then we can probably save space by encoding.

For example, suppose that the characters appear with the following frequencies, and following codes:

	a	b	c	d	e	f	g	h
Frequency	45K	13K	12K	16K	9K	5K	0K	0K
Fixed-length code	000	001	010	011	100	101	110	111
Variable-length code	0	101	100	111	1101	1100	—	—

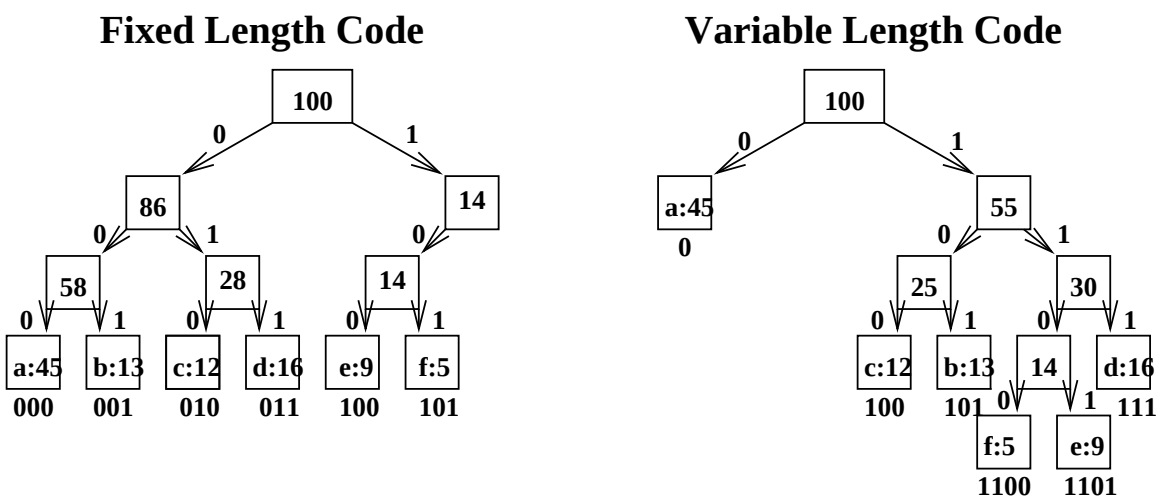
Then the variable-length coded version will take not 300K bits but $45\text{K} \cdot 1 + 13\text{K} \cdot 3 + 12\text{K} \cdot 3 + 16\text{K} \cdot 3 + 9\text{K} \cdot 4 + 5\text{K} \cdot 4 = 224\text{K}$ bits to store, a 25% saving. In fact this is the optimal way to encode the 6 characters present, as we shall see.

We will consider only codes in which no code is a prefix of any other code; such codes are called *prefix codes* (though perhaps they should be called prefix-free codes). The attraction of such codes is that it is easy to encode and decode data. To encode, we need only concatenate the codes of consecutive characters in the message. So for example "face" is encoded as "110001001101". To decode, we have to decide where each code begins and ends, since they are no longer all the same length. But this is easy, since, no codes share a prefix. This means we need only scan the input string from left to right, and as soon as we recognize a code, we can print the corresponding character and start looking for the next code. In the above case, the only code that begins with "1100..." or a prefix is "f", so we can print "f" and start decoding "0100...", get "a", etc.

To see why the no-common prefix property is essential, suppose that we tried to encode "e" with the shorter code "110", and tried to decode "1100"; we could not tell whether this

represented "ea" or "f". (Furthermore, one can show that one cannot compress any better with a non-prefix code, although we will not show this here.)

We can represent the decoding algorithm by a binary tree, where each edge represents either 0 or 1, and each leaf corresponds to the sequence of 0s and 1s traversed to reach it, ie a particular code. Since no prefix is shared, all legal codes are at the leaves, and decoding a string means following edges, according to the sequence of 0s and 1s in the string, until a leaf is reached. The tree for the above code is shown on the right below, along with a tree on the left for the fixed length 3-bit code:



Each leaf is labeled by the character it represents (before the colon), as well as the frequency with which it appears in the text (after the colon, in 1000s). Each internal node is labeled by the frequency with which all leaf nodes under it appear in the text (ie the sum of their frequencies). The bit string representing each character is also shown beneath each leaf.

We will denote the set of all characters by X, an arbitrary character by $x \in X$, the frequency with with x appears by $f(x)$, and its depth in the tree by $d(x)$. Note that $d(x)$ is the number of bits in the code for x.

Given a tree T representing a prefix code, it is easy to compute the number of bits needed to represent a file with the given frequencies $f(x)$: $B(T) = \sum_{x \in X} f(x)d(x)$, which we call the *cost of* T.

The greedy algorithm for computing the optimal Huffman coding tree T given the character frequencies $f(x)$ is as follows. It starts with a forest of one-node trees representing each $x \in X$, and merges them in a greedy style reminiscent of Prim's MST algorithm, using a priority queue Q, sorted by the *smallest* frequency:

procedure Huffman(X, $f(.)$)
$n = |X|$, the number of characters
for all $x \in X$, enqueue($(x, f(x)), Q$)
for $i = 1$ to n
    allocate a new tree node z
    $left_child$ =deletemin(Q)

$right_child$ =deletemin(Q)
$f(z) = f(left_child) + f(right_child)$
Make $left_child$ and $right_child$ the children of z
enqueue($(z, f(z)), Q$)

The cost of this algorithm is clearly the cost of $2n$ deletemins and n enqueues onto the priority queue Q. Assuming Q is implemented with a binary heap, so that these operations cost $O(\log n)$, the whole algorithm costs $O(n \log n)$, or as much as sorting.

Here is why it works, i.e., produces the tree T minimizing $B(T)$ over all possible trees. We will use induction on the number of characters in X. When $|X| = 2$, the optimal tree clearly has two leaves, corresponding to strings 0 and 1, which is what the algorithm constructs. Now suppose $|X| > 2$. The first thing the algorithm does is make the two lowest-frequency characters (call them x and y) into leaf nodes, create a new node z with frequency $f(z)$ equal to the sum of their frequencies, and apply the algorithm to the new set $\bar{X} = X - \{x, y\} \cup \{z\}$, which has $|\bar{X}| = |X| - 1$ characters, so we can apply induction. Thus, we can assume that the algorithm builds an optimal tree $\bar{T}$ for $\bar{X}$.

The trick is to show that T, gotten by adding x and y as left and right children to z in $\bar{T}$, is also optimal. We do this by contradiction: suppose there is a better tree T' for X, i.e., with a cost $B(T') < B(T)$. Then we will show there is a better tree $\bar{T}'$ for $\bar{X}$, ie with $B(\bar{T}') < B(\bar{T})$, contradicting the optimality of $\bar{T}$.

First note that

$$\begin{aligned} B(T) &= B(\bar{T}) - f(z)d(z) + f(x)d(x) + f(y)d(y) \\ &= B(\bar{T}) - (f(x) + f(y))(d(x) - 1) + f(x)d(x) + f(y)d(x) \\ &= B(\bar{T}) + f(x) + f(y) \ . \end{aligned}$$

Similarly, if x and y are siblings in T', we can build the tree $\bar{T}'$ by removing x and y from T' and making their parent a leaf. The same argument shows that $B(T') = B(\bar{T}') + f(x) + f(y)$. Since this is less than $B(T) = B(\bar{T}) + f(x) + f(y)$, we get $B(\bar{T}') < B(\bar{T})$, the contradiction we wanted.

When x and y are *not* siblings in T', we have to work harder: we will show that there is *another* optimal tree T'' where they *are* siblings, so we can apply the above argument with T'' in place of T'. To construct T'' from T', let b and c be the two siblings of maximum depth $d(b) = d(c)$ in T'. Since all leaf nodes have siblings by construction, we have $d(b) = d(c) \geq d(z)$ for any other leaf z. Suppose without loss of generality that $f(x) \leq f(y)$ and $f(b) \leq f(c)$ (if this is not true, swap the labels x and y, or b and c). Then $f(x) \leq f(b)$ and $f(y) \leq f(c)$. Then let T'' be the tree with b and x swapped, and c and y swapped. Clearly x and y are siblings in T''. We need to show that $B(T'') \leq B(T')$; since we know that $B(T')$ is minimal, this will mean that $B(T'') = B(T')$ is minimal as well. We compute as follows (the depths $d(.)$ refer to T'):

$$\begin{aligned} B(T'') &= B(T') + [-f(b)d(b) - f(x)d(x) + f(b)d(x) + f(x)d(b)] \\ &\quad +[-[f(c)d(c) - f(y)d(y) + f(c)d(y) + f(y)d(c)] \\ &= B(T') - [(f(b) - f(x)) \cdot (d(b) - d(x))] - [(f(c) - f(y)) \cdot (d(c) - d(y))] \\ &\leq B(T') \end{aligned}$$

since all the quantities in parenthesis are nonnegative. This completes the construction of T'' and the proof.