SIENA*college*
Computer Science

Computer Science 385
Analysis of Algorithms
Siena College
Spring 2011

Topic Notes: Huffman Codes

We will now briefly consider another greedy algorithm concerned with the generation of encodings.

The problem of *coding* is assignment of bit strings to alphabet characters. *Codewords* are the bit strings assigned for characters of alphabet.

We can categorize codes as one of:

1. *fixed-length encoding* (*e.g.*, ASCII)
2. *variable-length encoding* (*e.g.*, Morse code)

The idea here is to reduce the amount of space needed to store a string of characters. Usually, we store characters with 8 bits each, meaning we can store up to $2^8 = 256$ different characters.

However, many strings don't use that many different characters. If we had a string that used only 12 unique characters, we could define patterns of 4 bits each to represent them and save half of the space.

The idea behind variable length encoding is that we can do even better if we use short strings of bits to represent frequently seen characters and infrequent characters with longer strings. This results in a smaller total number of bits needed to encode a message.

To do this, we need to come up with a code and a way to translate text into code and then back again.

But...these variable length codes introduce a problem. If each character's code can have a different length, how do we know when the code for one character has ended and the next has begun?

In Morse code, how can we tell the if the sequence "dot dash dash" is supposed to represent "AT", "ETT", "EM" or just the one character "W"?

This is possible because Morse code is not a binary code at all – it does have dots and dashes (which are one and three time units, respectively, of the sound), but it also has pauses of varying length to separate the individual dots and dashes (a period of silence equal in duration to the sound of a "dot"), to separate letters (silence for the duration of a dash), and to separate words (silence for the duration of 7 dots).

A strictly binary code cannot have these other "markers" to separate letters or words. Therefore, we would construct a *prefix-free code*, one where no codeword is a prefix of another codeword.

This leads to an interesting problem: if the frequencies of the character occurrences for the string to be encoded are known, what is the best binary prefix-free code?

Consider this procedure to generate a translation system, known as a *Huffman coding*.

Count the number of each character in the string to represent and create a single-node binary tree with that character and its count as the value. Repeatedly take the smallest two trees in the collection and combine them to a new tree which has the two trees as subtrees and label the root with the sum of their counts. Continue combining trees (both the original one-element trees and the trees created) in this manner until a single tree remains.

Consider the phrase:

```
no... try not... do... or do not... there is no try...
```

We count the letters up:

n=4, o=7, .=15, ␣=10, t=5, r=4, y=2, d=2, h=1, e=2, i=1, s=1

and build the tree.

Once we have that, we can use it to construct our encoded (compressed) string.

To decode, we just trace the bit patterns through the tree. When we encounter a leaf, we know the next letter. We then start tracing at the root again.

Note that the construction is a greedy procedure: we simply take the tree from our collection that has the smallest number of characters represented.