CS 624
Lecture 9: A Greedy Algorithm: Huffman Encoding

1 Data compression and prefix codes

The term “encoding” is used in several senses. Sometimes it is used to mean some way of hiding the contents of a message. That’s not what it means here. What we are really concerned with there is data compression. That is, how can we take a message and encode it in a way that it takes up as little space as possible? (That way, for instance, it would be cheaper to transmit, cheaper to store on disk, and so on.)

There are many ways that have been proposed to do this kind of thing. The text gives an interesting example: suppose we have a “message” or data file composed of 100,000 characters. And say there are only 6 different characters that are actually used: $\{a, b, c, d, e, f\}$. One way to encode this as a binary file is to use a *fixed-length* encoding:

character	code
a	000
b	001
c	010
d	011
e	100
f	101

Note that we need three bits for each character. And so the entire message will take 300,000 bits to encode. Can we do better?

Suppose we know that some characters are used more often than others. Suppose, for instance, that the number of characters actually used looks like this:

character	times used
a	45,000
b	13,000
c	12,000
d	16,000
e	9.000
f	5,000

Then a fixed-length encoding is probably not what we want. In fact, it would seem that we could do better if a had a shorter code than f, for instance. In other words, we look for a *variable-length* encoding.

Of course this leads immediately to a problem: if the different characters can be represented by code words of different lengths, how do we know when we have gotten to the end of a code word?

One good way to deal with this is to restrict ourselves to what are called *prefix* codes. (As the book points out, the term really should be *prefix-free* codes, and sometimes you do see this, but the term *prefix* codes is unfortunately also standard.)

A prefix code is a code in which no code word is the prefix of another one. So you automatically know when you have reached the end of a code word.

Here is a prefix code for the case we have been considering. Note that it really is a prefix code.

character	times used	code
a	45,000	0
b	13,000	101
c	12,000	100
d	16,000	111
e	9.000	1101
f	5,000	1100

The total size of the encoded message is now

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224,000 \text{ bits}$$

which is a significant improvement, even though some of the code words are actually longer this time.

Actually, it's customary to reason, not in terms of the actual number of occurrences of each character in the message, but in terms of their *frequency*, which is just the number of occurrences divided by the total number of characters in the message. You can also think of the frequency of a as the probability that any particular character is an a. So we would write this as

character	frequency	code
a	.45	0
b	.13	101
c	.12	100
d	.16	111
e	.09	1101
f	.05	1100

and

$$1(.45) + 3(.13) + 3(.12) + 3(.16) + 4(.09) + 4(.05) = 2.24$$

is the expected number (or "average" number) of bits per character—as opposed to 3 bits per character in our fixed-length encoding.

We see that we can measure the efficiency of a code by the expected number of bits per character. Suppose we let C be the set of characters. If we let x be a variable that runs over the set of characters in C, and if $f(x)$ is the frequency of the character x, and if $\mathit{length}(x)$ is the length of the code word corresponding to x, then the average number of bits per character will be simply

$$\sum_{x \in C} f(x) \cdot \mathit{length}(x)$$

It's also true, by the way, that

$$\sum_{x \in C} f(x) = 1$$

It turns out, however, that we don't actually need this fact in what we are doing here. Just think of the values of the function f as weights.

Our problem, then is—given the set C and the frequency function f, to come up with a prefix encoding that minimizes this value.

2 Decoding

Before we produce an algorithm, let us consider the problem of *decoding*. This can be represented as a binary tree. For instance, Figure 1 shows the trees for the original fixed-length code and the prefix code we produced above.

The decoding tree really is another way of defining the encoding as well, and it's a very useful one for us, because it gives us important structural information about the encoding that allows us to reason about it.

For instance, note that the depth of a leaf in the tree is just the length of the code word for that character. Thus, if we let $d_T(x)$ be the depth of a leaf node corresponding to the character x in the

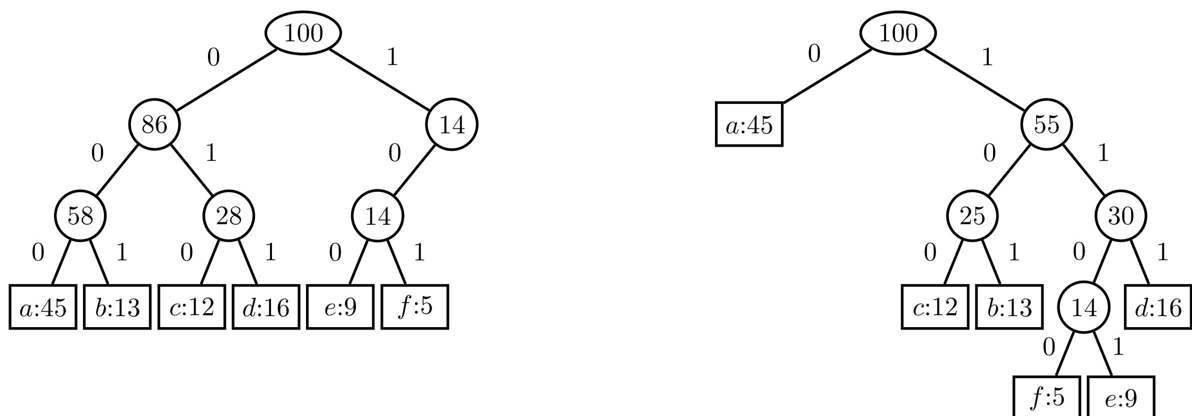

Figure 1: Two encoding trees. Each leaf is labeled with a character and its frequency (as a percent) of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree.

tree T, then the average cost AC per character in the encoding scheme defined by the tree T is

$$AC(T) = \sum_{x \in C} f(x) d_T(x) \tag{1}$$

Also note that in the prefix encoding tree, each internal node has exactly two children. (Another way to say this is that each node has either 2 or 0 children.) This actually is an important property of such trees:

2.1 Exercise *Show that if a tree representing a prefix code contains an internal node with 1 child, then there is a tree with a more efficient prefix code.*

3 Approaches to solving the problem

3.1 Exhaustive search

Given what we know so far, one way we could solve the problem of finding an optimal encoding would be to enumerate all possible prefix trees and find the one with the smallest average cost per character. Without performing an exact analysis, it's pretty obvious that the cost of this algorithm would be exponential in the number of characters, and therefore completely useless.

3.2 A recursive method

So we really need some more information about the properties of prefix codes in order to do better. Here is an important fact:

3.1 Lemma *If T is the tree corresponding to an optimal prefix encoding, and if T_L and T_R are its left and right subtrees, respectively, then T_L and T_R are also trees corresponding to optimal prefix encodings.*

PROOF. Let us say that C_L is the set of characters that are leaf nodes in T_L and similarly for C_R and T_R.

If $x \in C_L$, then certainly $d_{T_L}(x) = d_T(x) - 1$, and the same is true for C_R and T_R. Therefore we can see from our basic cost formula (1) that

$$\begin{aligned} AC(T) &= \sum_{x \in C} f(x) d_T(x) \\ &= \sum_{x \in C_L} f(x)\big(d_{T_L}(x) + 1\big) + \sum_{x \in C_R} f(x)\big(d_{T_R}(x) + 1\big) \\ &= \sum_{x \in C_L} f(x) d_{T_L}(x) + \sum_{x \in C_R} f(x) d_{T_R}(x) + \sum_{x \in C} f(x) \end{aligned}$$

Now if T_R was not an optimal encoding tree, then we could replace it by a more efficient one (with the same leaves and the same frequencies), and this would show in turn that T could not have been optimal. And that's a contradiction. □

3.2 Corollary *If T is the tree corresponding to an optimal prefix encoding, then every subtree of T also corresponds to an optimal prefix encoding.*

PROOF. This follows immediately by induction. □

This lemma expresses the fact that the problem of finding an optimal prefix code has the *optimal substructure property*. This means that we could write a recursive algorithm for it. It would go something like this:

Start with a worklist consisting of n trees, each tree consisting of exactly 1 character. From these trees construct other trees bottom-up and add them to the worklist. As each new tree is constructed, check the worklist to see if a tree with the same leaves is in it. Keep the tree with the smallest cost in the worklist and remove any others with the same set of leaves. At the end of this process there will be one tree in the worklist that contains all the characters in C as leaves, and that tree represents an optimal encoding.

This algorithm will definitely give the correct answer, but again it is quite inefficient, although it is better than complete exhaustive search.

3.3 A greedy method—Huffman's algorithm

The optimal substructure property should remind us of dynamic programming. And if there were also an *overlapping subproblems* property of this problem, we could try such a solution. But in fact, we have something even better: We don't actually have to form all possible trees on the way up and check them all. We actually can tell at each step exactly which tree to form.

3.3 Lemma *Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.*

PROOF. To prove this, suppose that the tree T represents an optimal prefix code for our problem. If x and y are sibling nodes of greatest depth, then we are done. Otherwise, suppose that p and q are sibling nodes of greatest depth. We will exchange x and p, and we will also exchange y and q.

Now we know that

$$\begin{aligned} d_T(x) &\le d_T(p) \\ d_T(y) &\le d_T(q) \\ f(x) &\le f(p) \\ f(y) &\le f(q) \end{aligned}$$

Let us say that the tree T, after these two switches, is turned into the tree T'. Then we have

$$\begin{aligned} d_{T'}(x) &= d_T(p) \\ d_{T'}(p) &= d_T(x) \\ d_{T'}(y) &= d_T(q) \\ d_{T'}(q) &= d_T(y) \end{aligned}$$

and so

$$\begin{aligned} AC(T') - AC(T) &= \sum_{z\in C} f(z)\big(d_{T'}(z) - d_T(z)\big) \\ &= f(p)\big(d_{T'}(p) - d_T(p)\big) + f(x)\big(d_{T'}(x) - d_T(x)\big) + f(q)\big(d_{T'}(q) - d_T(q)\big) + f(y)\big(d_{T'}(y) - d_T(y)\big) \\ &= f(p)\big(d_T(x) - d_T(p)\big) + f(x)\big(d_T(p) - d_T(x)\big) + f(q)\big(d_T(y) - d_T(q)\big) + f(y)\big(d_T(q) - d_T(y)\big) \\ &= \big(f(p) - f(x)\big)\big(d_T(x) - d_T(p)\big) + \big(f(q) - f(y)\big)\big(d_T(y) - d_T(q)\big) \\ &\le 0 \end{aligned}$$

so $AC(T') \le AC(T)$, which shows that T was not an optimal tree to begin with, and this is a contradiction. □

This means that we can start out with our initial worklist, and we can take two nodes of smallest frequency and build a tree from them (in which they are the two leaves). Then we can delete those two nodes from the worklist, because we know that they will definitely be part of the little tree we have just constructed—we will never have to look at them again.

Then we can continue. By exactly the same argument, we can take the two elements of the worklist that are now of smallest cost, and build a little tree from them, and then throw them away. When we are done, we will have the tree we are looking for.

The then is the algorithm: We will keep a minimum-priority queue Q of subtrees. Q initially consists of the n characters. The priority of any element in Q will be the cost of that subtree.

```
HUFFMAN(C)
  n ← |C|
  Q ← C
  for i ← 1 to n − 1 do
    allocate a new node z
    left[z] ← EXTRACT-MIN(Q)
    right[z] ← EXTRACT-MIN(Q)
    f[z] ← f[x] + f[y]
    INSERT(Q.z)
  return EXTRACT-MIN(Q) ▷ Return the root of the tree.
```

Thus, this algorithm works even better than a dynamic programming algorithm: we don't have to memoize intermediate results for later use. We know exactly at each step what we need to do. This is called a “greedy” algorithm because we chose the locally best solution at each step. In effect, we act as if we were “greedy”. What is is the best at each step is guaranteed (in this case) to turn to out to be the best overall.