CMSC 451: Lecture 6
Greedy Algorithms: Huffman Coding
Thursday, Sep 14, 2017

Reading: Sect. 4.8 in KT and Sect. 5.2 in DPV.

Greedy Algorithms: In an *optimization problem*, we are given an input and asked to compute a structure, subject to various constraints, in a manner that either minimizes cost or maximizes profit. Such problems arise in many applications of science and engineering. Given an optimization problem, we are often faced with the question of whether the problem can be solved efficiently (as opposed to a brute-force enumeration of all possible solutions), and if so, what approach should be used to compute the optimal solution?

In many optimization algorithms a series of selections need to be made. A simple design technique for optimization problems is based on a *greedy* approach, that builds up a solution by selecting the best alternative in each step, until the entire solution is constructed. When applicable, this method can lead to very simple and efficient algorithms. (Unfortunately, it does not always lead to optimal solutions.)

Today, we will consider one of the most well-known examples of a greedy algorithm, the construction of Huffman codes.

Huffman Codes: Huffman codes provide a method of encoding data efficiently. Normally when characters are coded using standard codes like ASCII or the Unicode, each character is represented by a fixed-length *codeword* of bits (e.g., 8 or 16 bits per character). Fixed-length codes are popular, because its is very easy to break a string up into its individual characters, and to access individual characters and substrings by direct indexing. However, fixed-length codes may not be the most efficient from the perspective of minimizing the total quantity of data.

Consider the following example. Suppose that we want to encode strings over the (rather limited) 4-character alphabet $C = \{a, b, c, d\}$. We could use the following fixed-length code:

Character	a	b	c	d
Fixed-Length Codeword	00	01	10	11

A string such as "abacdaacac" would be encoded by replacing each of its characters by the corresponding binary codeword.

a	b	a	c	d	a	a	c	a	c
00	01	00	10	11	00	00	10	00	10

The final 20-character binary string would be "00010010110000100010".

Now, suppose that you knew the relative probabilities of characters in advance. (This might happen by analyzing many strings over a long period of time. In applications like data compression, where you want to encode one file, you can just scan the file and determine the exact frequencies of all the characters.) You can use this knowledge to encode strings differently. Frequently occurring characters are encoded using fewer bits and less frequent characters are encoded using more bits. For example, suppose that characters are expected to occur with the following probabilities. We could design a *variable-length code* which would do a better job.

Character	a	b	c	d
Probability	0.60	0.05	0.30	0.05
Variable-Length Codeword	0	110	10	111

Notice that there is no requirement that the alphabetical order of character correspond to any sort of ordering applied to the codewords. Now, the same string would be encoded as follows.

a	b	a	c	d	a	a	c	a	c
0	110	0	10	111	0	0	10	0	10

Thus, the resulting 17-character string would be "01100101110010010". Thus, we have achieved a savings of 3 characters, by using this alternative code. More generally, what would be the expected savings for a string of length n? For the 2-bit fixed-length code, the length of the encoded string is just $2n$ bits. For the variable-length code, the expected length of a single encoded character is equal to the sum of code lengths times the respective probabilities of their occurrences. The expected encoded string length is just n times the expected encoded character length.

$$n(0.60 \cdot 1 + 0.05 \cdot 3 + 0.30 \cdot 2 + 0.05 \cdot 3) \;=\; n(0.60 + 0.15 + 0.60 + 0.15) \;=\; 1.5n.$$

Thus, this would represent a 25% savings in expected encoding length. (Of course, we would also need to consider the cost of transmitting the code book itself, but typically the code book is much smaller than the text being transmitted.) The question that we will consider today is how to form the *best code*, assuming that the probabilities of character occurrences are known.

Prefix Codes: One issue that we didn't consider in the example above is whether we will be able to *decode* the string, once encoded. In fact, this code was chosen quite carefully. Suppose that instead of coding the character "a" as 0, we had encoded it as 1. Now, the encoded string "111" is ambiguous. It might be "d" and it might be "aaa". How can we avoid this sort of ambiguity? You might suggest that we add separation markers between the encoded characters, but this will tend to lengthen the encoding, which is undesirable. Instead, we would like the code to have the property that it can be uniquely decoded.

Note that in both the variable-length codes given in the example above no codeword is a *prefix* of another. This turns out to be critical. Observe that if two codewords did share a common prefix, e.g. a $\rightarrow$ 001 and b $\rightarrow$ 00101, then when we see 00101 . . . how do we know whether the first character of the encoded message is "a" or "b". Conversely, if no codeword is a prefix of any other, then as soon as we see a codeword appearing as a prefix in the encoded text, then we know that we may decode this without fear of it matching some longer codeword. Thus we have the following definition.

Prefix Code: Mapping of codewords to characters so that no codeword is a prefix of another.

Observe that any binary prefix coding can be described by a binary tree in which the codewords are the leaves of the tree, and where a left branch means "0" and a right branch means "1". The length of a codeword is just its depth in the tree. The code given earlier is a prefix code, and its corresponding tree is shown in Fig. 1.

Character	a	b	c	d
Codeword	0	110	10	111

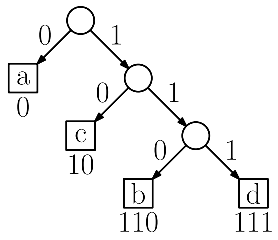

Fig. 1: A tree-representation of a prefix code.

Decoding a prefix code is simple. We just traverse the tree from root to leaf, letting the input character tell us which branch to take. On reaching a leaf, we output the corresponding character, and return to the root to continue the process.

Expected encoding length: Once we know the probabilities of the various characters, we can determine the total length of the encoded text. Let $p(x)$ denote the probability of seeing character x, and let $d_T(x)$ denote the length of the codeword (depth in the tree) relative to some prefix tree T. The expected number of bits needed to encode a single character is given in the following formula:

$$\boxed{B(T) = \sum_{x \in C} p(x) d_T(x).}$$

definition of the expected value

This suggests the following problem:

Optimal Code Generation: Given an alphabet C and the probabilities $p(x)$ of occurrence for each character $x \in C$, compute a prefix code T that minimizes the expected length of the encoded bit-string, $B(T)$.

There is an elegant greedy algorithm for finding such a code. It was invented in the 1950's by David Huffman, and is called a *Huffman code.* (While the algorithm is simple, it was not obvious. Huffman was a student at the time, and his professors, Robert Fano and Claude Shannon, two very eminent researchers, had developed their own algorithm, which as suboptimal.)

By the way, Huffman coding was used for many years by the Unix utility `pack` for file compression. Later it was discovered that there are better compression methods. For example, `gzip` is based on a more sophisticated method called the *Lempel-Ziv coding* (in the form of an algorithm called *LZ77*), and `bzip2` is based on combining the *Burrows-Wheeler transformation* (an extremely cool invention!) with run-length encoding, and Huffman coding.

Huffman's Algorithm: Here is the intuition behind the algorithm. Recall that we are given the occurrence probabilities for the characters. We are going to build the tree up from the leaf level. We will take two characters x and y, and "merge" them into a single *meta-character* called z, which then replaces x and y in the alphabet. The character z will have a probability equal to the sum of x and y's probabilities. Then we continue recursively building the code on the new alphabet, which has one fewer character. When the process is completed, we know the code for z, say 010. Then, we append a 0 and 1 to this codeword, given 0100 for x and 0101 for y.

Another way to think of this, is that we merge x and y as the left and right children of a root node called z. Then the subtree for z replaces x and y in the list of characters. We repeat this process until only one meta-character remains. The resulting tree is the final prefix tree. Since x and y will appear at the bottom of the tree, it seem most logical to select the two characters with the smallest probabilities to perform the operation on. The result is Huffman's algorithm. It is illustrated in Fig. 2.

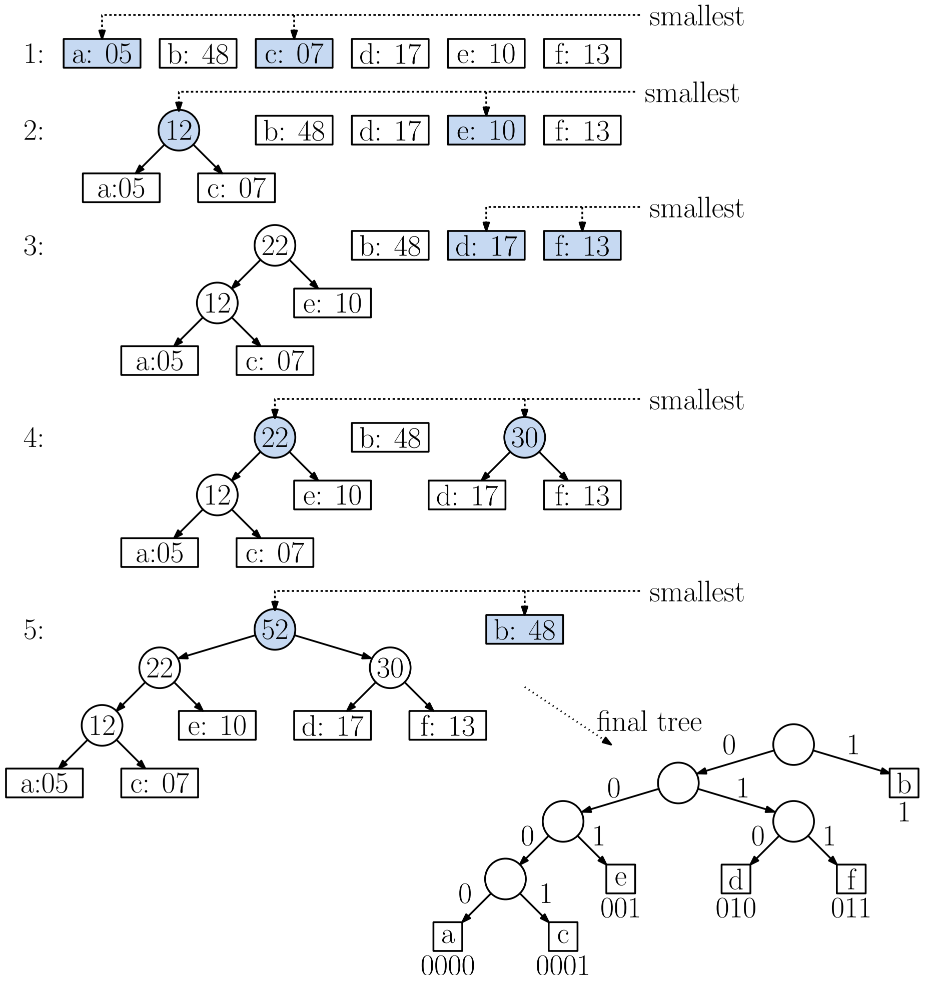

Fig. 2: Huffman's Algorithm.

The pseudocode for Huffman's algorithm is given below. Let C denote the set of characters, and let $n = |C|$. Each character $x \in C$ is associated with an occurrence probability prob$[x]$.

O(n) for fib heap

Initially, the characters are all stored in a *priority queue* Q. Recall that this data structure can be built initially in $O(n)$ time, and we can extract the element with the smallest key in $O(\log n)$ time and insert a new element in $O(\log n)$ time. The objects in Q are sorted by probability. Note that with each execution of the for-loop, the number of items in the queue decreases by one. So, after $n-1$ iterations, there is exactly one element left in the queue, and this is the root of the final prefix code tree.

Huffman's Algorithm

```
huffman(C, prob) {                      // C = chars, prob = probabilities
    for each (x in C) {
        add x to Q sorted by prob[x]      // add all to priority queue
    }
    for (i = 1 to |C| - 1) {              // repeat until only 1 item in queue
        z = new internal tree node
        left[z]  = x = extract-min from Q // extract min probabilities
        right[z] = y = extract-min from Q
        prob[z]  = prob[x] + prob[y]      // z's probability is their sum
        insert z into Q                   // z replaces x and y
    }
    return the last element left in Q as the root
}
```

Correctness: The big question that remains is why is this algorithm correct, that is, does it compute the tree that minimizes the expected encoding length? Recall that the cost of any encoding tree T is $B(T) = \sum_x p(x)d_T(x)$. Our approach will be to show that any tree that differs from the one constructed by Huffman's algorithm can be converted into one that is equal to Huffman's tree without increasing its cost. This is done by identifying an appropriate place where the two solutions differ, modify the non-greedy solution so that it is a bit closer to the greedy solution, and showing that this modification can be done so that the cost does not increase. By repeating this, we will eventually modify any solution into the greedy solution in a manner that does not increase this cost. This implies that the greedy solution is has the minimum cost.

Our approach is based a few observations. First, observe that the Huffman tree is a *full binary tree*, meaning that every internal node has exactly two children. (It would never pay to have an internal node with only one child, since we could replace this node with its child without increasing the tree's cost.) So we may safely limit consideration to full binary trees. Our next observation (proved below) is that in any optimal code tree, the two characters with the lowest probabilities will be siblings at the maximum depth in the tree. Once we have this fact, we will merge these two characters into a single meta-character whose probability is the sum of their individual probabilities. As a result, we will now have one less character in our alphabet. This will allow us to apply induction to the remaining $n-1$ characters.

Let's first prove the above assertion that the two characters of lowest probability may be assumed to be siblings at the lowest level of the tree.

Claim 1: Consider the two characters, x and y with the smallest probabilities. Then there is an optimal code tree in which these two characters are siblings at the maximum depth in the tree.

Proof: Let T be any optimal prefix code tree, and let b and c be two siblings at the maximum depth of the tree. (There may be many such siblings, and if so pick any such pair.) If $\{x, y\} = \{b, c\}$ we are done. Otherwise, from the fact that x and y have the lowest probabilities, we may label the nodes such that $p(b) \leq p(c)$ and $p(x) \leq p(y)$.

Now, since x and y have the two smallest probabilities it follows that $p(x) \leq p(b)$ and $p(y) \leq p(c)$. (In both cases they may be equal.) Because b and c are at the deepest level of the tree we know that $d_T(b) \geq d_T(x)$ and $d_T(c) \geq d_T(y)$. (Again, they may be equal.) Thus, we have $p(b) - p(x) \geq 0$ and $d_T(b) - d_T(x) \geq 0$, and hence their product is nonnegative. Now, suppose that we switch the positions of x and b in the tree, resulting in a new tree T' (see Fig. 3).

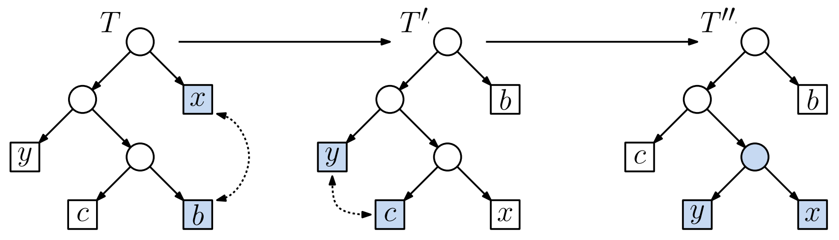

Fig. 3: Showing that the lowest probability nodes are siblings at the tree's lowest level.

Next let us see how the cost changes as we go from T to T'. Almost all the nodes contribute the same to the expected cost in both trees. The only exceptions are nodes x and b. By subtracting the old contributions of these nodes and adding in the new contributions we have

$$\begin{aligned} B(T') &= B(T) - (\text{old cost for } b \text{ and } x) + (\text{new cost for } b \text{ and } x) \\ &= B(T) - (p(x)d_T(x) + p(b)d_T(b)) + (p(x)d_T(b) + p(b)d_T(x)). \end{aligned}$$

With a little algebraic manipulation we obtain

$$\begin{aligned} B(T') &= B(T) + p(x)(d_T(b) - d_T(x)) - p(b)(d_T(b) - d_T(x)) \\ &= B(T) - (p(b) - p(x))(d_T(b) - d_T(x)) \\ &\leq B(T), \end{aligned}$$

where the last step follows because $(p(b) - p(x))(d_T(b) - d_T(x)) \geq 0$. Thus the cost does not increase. (Given our assumption that T was already optimal, it certainly cannot decrease either, since otherwise we would have a contradiction.) Since T was an optimal tree, T' is also an optimal tree.

By a similar argument, we can switch y with c to obtain a new tree T''. Again, the same sort of argument implies that T'' is also optimal. The final tree T'' satisfies the statement of the claim.

The above claim applies to just one pair of nodes, those with the lowest probabilities. To show that the *entire* Huffman tree is optimal, we need to extend this argument. We will do this by induction. In order to reduce from n characters to $n-1$, we will do the same reduction that Huffman's algorithm does; namely we will *merge* characters x and y into a new meta-character z, whose probability is the sum of the probabilities of x and y.

Claim 2: Let T_n be any prefix-code tree that satisfies the property of Claim 1 (lowest probability symbols x and y are siblings at the deepest level). Let T_{n-1} be the tree that results by replacing these two nodes and their parent with a single leaf node z of probability $p(z) = p(x) + p(y)$. Then $B(T_n) = B(T_{n-1}) + p(z)$. $* d_T(z)$?

Proof: Let d denote the depths of x and y in T_n. Clearly, z is at depth $d-1$ in T_{n-1} (see Fig. 4).

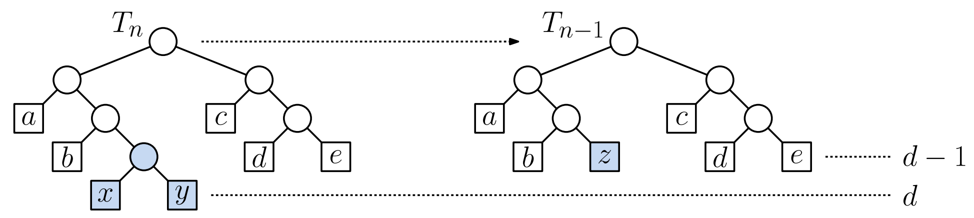

Fig. 4: Proving the correctness of Huffman's algorithm.

Because z replaces x and y the costs of the two trees satisfies

$$\begin{aligned}
B(T_n) &= B(T_{n-1}) - (z\text{'s cost in } B(T_{n-1})) + (x \text{ and } y\text{'s costs in } B(T_n)) \\
&= B(T_{n-1}) - p(z)(d-1) + (p(x)\,d + p(y)\,d) \\
&= B(T_{n-1}) - p(z)\,(d-1) + p(z)\,d \\
&= B(T_{n-1}) + p(z).
\end{aligned}$$

Note that the cost of trees T_n and T_{n-1} differ only by the fixed term $p(z)$, which does not depend on the tree's structure. Therefore (subject to this replacement), minimizing the cost of T_n is equivalent to minimizing the cost of T_{n-1}. This allows us to prove our main result.

Claim 3: Huffman's algorithm produces an optimal prefix code tree.

Proof: The proof is by induction on n, the number of characters. The basis case ($n = 1$) is trivial, since there is only one tree possible. If $n \geq 2$, then by Claim 1, we know that the two characters x and y of lowest probability are siblings at the deepest level of an optimal tree. Huffman's algorithm replaces these nodes by a character z whose probability is the sum of their probabilities. By induction, Huffman's algorithm computes the optimum tree over the resulting alphabet of $n-1$ symbols. Call it T_{n-1}. Replacing z with nodes x and y results in a tree T_n whose cost is higher by the fixed amount $p(z) = p(x) + p(y)$. Since T_{n-1} is optimal, and the cost of replacement does not depend on the tree's structure, T_n is also optimal.