In today's lecture we will be looking a bit more closely at the Greedy approach to designing algorithms. As we will see, sometimes it works, and sometimes even when it doesn't, it can provide a useful result.

Horn Formulae

A simple application of the greedy paradigm solves an important special case of the SAT problem. We have already seen that 2SAT can be solved in linear time. Now consider SAT instances where in each clause, there is at most one positive literal. Such formulae are called *Horn formulae*; for example, this is an instance:

$$(x \vee \overline{y} \vee \overline{z} \vee \overline{w}) \wedge (\overline{x} \vee \overline{y} \vee \overline{w}) \wedge (\overline{x} \vee \overline{z} \vee w) \wedge (\overline{x} \vee y) \wedge (x) \wedge (\overline{z}) \wedge (\overline{x} \vee \overline{y} \vee w).$$

Given a Horn formula, we can separate its clauses into two parts: the *pure negative clauses* (those without a positive literal) and the *implications* (those with a positive literal). We call clauses with a positive literal implications because they can be rewritten suggestively as implications; $(x \vee \overline{y} \vee \overline{z} \vee \overline{w})$ is equivalent to $(y \wedge z \wedge w) \rightarrow x$. Note the trivial clause (x) can be thought of as a trivial implication $\rightarrow x$. Hence, in the example above, we have the implications

$$(y \wedge z \wedge w \rightarrow x), (x \wedge z \rightarrow w), (x \rightarrow y), (\rightarrow x), (x \wedge y \rightarrow w)$$

and these two pure negative clauses

$$(\overline{x} \vee \overline{y} \vee \overline{w}), (\overline{z}).$$

We can now develop a greedy algorithm. The idea behind the algorithm is that we start with all variables set to false, and we only set variables to T if an implication forces us to. Recall that an implication is not satisfied if all variables to the left of the arrow are true and the one to the right is false. This algorithm is greedy, in the sense that it (greedily) tries to ensure the pure negative clauses are satisfied, and only changes a variable if absolutely forced.

Algorithm Greedy-Horn(ϕ: CNF formula with at most one positive literal per clause)

Start with the truth assignment $t :=$FFF$\cdots$F

while there is an implication that is not satisfied do

    make the implied variable T in t

if all pure negatives are satisfied then return t

    else return "ϕ is unsatisfiable"

Once we have the proposed truth assignment, we look at the pure negatives. If there is a pure negative clause that is not satisfied by the proposed truth assignment, the formula cannot be satisfied. This follows from the fact that all the pure negative clauses will be satisfied if any of their variables are set to F. If such a clause is unsatisfied, all of its variables must be set to T. But we only set a variable to T if we are forced to by the implications. If all the pure negative clauses are satisfied, then we have found a truth assignment.

On the example above, Greedy-Horn first flips x to true, forced by the implication $\rightarrow x$. Then y gets forced to true (from $x \rightarrow y$), and similarly w is forced to true. (Why?) Looking at the pure negative clauses, we find that the first is not satisfied, and hence we conclude the original formula had no truth assignment.

Exercise: Show that the Horn-greedy algorithm can be implemented in linear time in the length of the formula (i.e., the total number of appearances of all literals).

Huffman Coding

Suppose that you must store the map of a chromosome which consists of a sequence of 130 million symbols of the form A, C, G, or T. To store the sequence efficiently, you represent each character with just 2 bits: A as 00, C as 01, G as 10, and T as 11. Such a representation is called an *encoding*. With this encoding, the sequence requires 260 megabits to store.

Suppose, however, that you know that some symbols appear more frequently than others. For example, suppose A appears 70 million times, C 3 million times, G 20 million times, and T 37 million times. In this case it seems wasteful to use two bits to represent each A. Perhaps a more elaborate encoding assigning a shorter string to A could save space.

We restrict ourselves to encodings that satisfy the *prefix property*: no assigned string is the prefix of another. This property allows us to avoid backtracking while decoding. For an example without the prefix property, suppose we represented A as 1 and C as 101. Then when we read a 1, we would not know whether it was an A or the beginning of a C! Clearly we would like to avoid such problems, so the prefix property is important.

You can picture an encoding with the prefix property as a binary tree. For example, the binary tree below corresponds to an optimal encoding in the above situation. (There can be more than one optimal encoding! Just flip the left and right hand sides of the tree.) Here a branch to the left represent a 0, and a branch to the right represents a 1. Therefore A is represented by 1, C by 001, G by 000, and T by 01. This encoding requires only 213 million bits – a 17% improvement over the balanced tree (the encoding 00,01,10,11). (This does not include the bits that might be necessary to store the form of the encoding!)
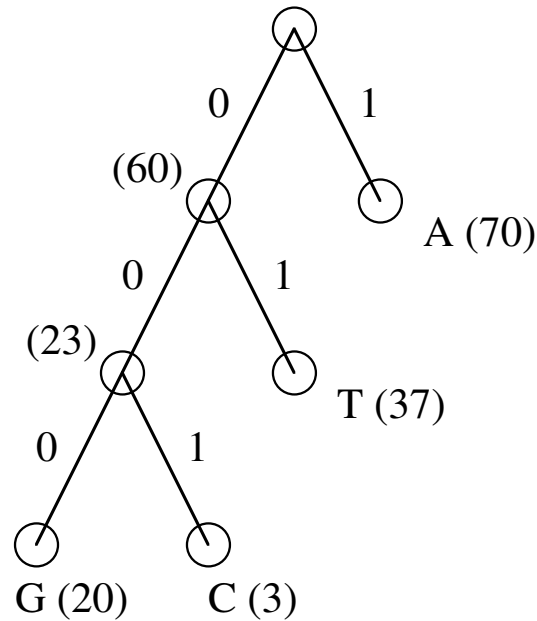
Figure 8.1: A Huffman tree.

Let us note some properties of the binary trees that represent encoding. The symbols must correspond to leaves; an internal node that represents a character would violate the prefix property. The code words are thus given by all root-to-leaf paths. All internal nodes must have exactly two children, as an internal node with only one child could be deleted to yield a better code. Hence if there are n leaves there are $n-1$ internal edges. Also, if we assign frequencies to the internal nodes, so that the frequencies of an internal node are the sums of the frequencies of the children, then the total length produced by the encoding is *the sum of the frequencies of all nodes except the root.* (A one line proof: each edge corresponds to a bit that is written as many times as the frequency of the node to which it leads.)

One final property allows us to determine how to build the tree: the two symbols with the smallest frequencies are together at the lowest level of the tree. Otherwise, we could improve the encoding by swapping a more frequently used character at the lowest level up. (This is not a full proof; feel free to complete one.)

This tells us how to construct the optimum tree greedily. Take the two symbols with the lowest frequency, delete them from the list of symbols, and replace them with a new meta-character; this new meta-character will lie directly above the two deleted symbols in the tree. Repeat this process until the whole tree is constructed.

We can prove by induction that this gives an optimal tree. It works for 2 symbols (base case). We also show that if it works for n letters, it must also work for $n+1$ letters. After deleting the two least frequent symbols and

replacing them with a meta-character, it as though we have just *n* symbols. this process yields an optimal tree for these *n* symbols (by the inductive hypothesis). Expanding the meta-character back into the two deleted nodes must now yield an optimal tree, since otherwise we could have found a better tree for the *n* symbols.
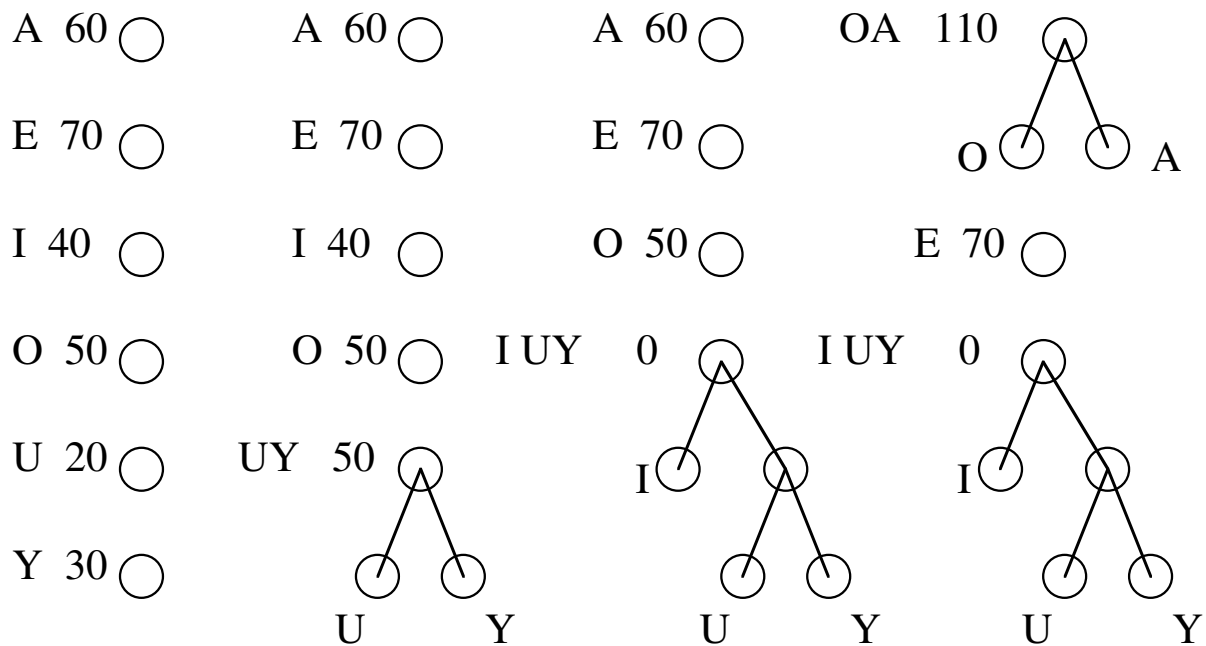

Figure 8.2: The first few steps of building a Huffman tree.

It is important to realize that when we say that a Huffman tree is optimal, this does not mean that it gives the best way to compress a string. It only says that we cannot do better by encoding one symbol at a time. By encoding frequently used blocks of letters (such as, in this section, the block "frequen") we can obtain much better encodings. (Note that finding the right blocks of letters can be quite complicated.) Given this, one might expect that Huffman coding is rarely used. In fact, many compression schemes use Huffman codes at some point as a basic building block. For example, image and video transmission often use Huffman encoding somewhere along the line.

Exercise: Find a Huffman compressor and another compressor, such as a gzip compressor. Test them on some files. Which compresses better?

It is straightforward to write code to generate the appropriate tree, and then use this tree to encode and decode messages. For encoding, we simply build a table with a codeword for each sybmol. To decode, we could read bits in one at a time, and walk down the tree in the appropriate manner. When we reach a leaf, we output the appropriate symbol and return to the top of the tree.

In practice, however, if we want to use Huffman coding, there are much faster ways to decode than to explicitly walk down the tree one bit at a time. Using an explicit tree is slow, for a variety of reasons. **Exercise:** Think about this.

One approach is to design a system that performs several steps at a time by reading several bits of input and determining what actions to take according to a big lookup table. For example, we could have a table that represents the information, "If you are currently at this point in the tree, and the next 8 bits are 00110101, then output AC and move to this point in the tree." This lookup table, which might be huge, encapsulates the information needed to handle eight bits at once. Since computers naturally handle eight bit blocks more easily than single bits, and because table lookups are faster than following pointers down a Huffman tree, substantial speed gains are possible. Notice that this gain in speed comes at the expense of the space required for the lookup table.

There are other solutions that work particularly well for very large dictionaries. For example, if you were using Huffman codes on a libraray of newspaper articles, you might treat each work as a symbol that can be encoded. In this case, you would have a lot of symbols! We will not go over these other methods here; a useful paper on the subject is "On the Implementation of Minimum-Redundancy Prefix Codes," by Moffat and Turpin. The key to keep in mind is that while thinking of decoding on the Huffman tree as happening one bit at a time is useful conceptually, good engineering would use more sophisticated methods to increase efficiency.

The Set Cover Problem

The inputs to the set cover problem are a finite set $X = \{x_1, \ldots, x_n\}$, and a collection of subsets $\mathcal{S}$ of X such that $\bigcup_{S \in \mathcal{S}} S = X$. The problem is to find the subcollection $\mathcal{T} \subseteq \mathcal{S}$ such that the sets of $\mathcal{T}$ cover X, that is

$$\bigcup_{T \in \mathcal{T}} T = X.$$

Notice that such a cover exists, since $\mathcal{S}$ is itself a cover.

The greedy heuristic suggests that we build a cover by repeatedly including the set in $\mathcal{S}$ that will cover the maximum number of as yet uncovered elements. In this case, the greedy heuristic does not yield an optimal solution. Interestingly, however, we can prove that the greedy solution is a good solution, in the sense that it is not too far from the optimal.

This is an example of an *approximation algorithm*. Loosely speaking, with an approximation algorithm, we settle for a result that is not the correct answer. Instead, however, we try to prove a guarantee on how close the algorithm is to the right answer. As we will see later in the course, sometimes this is the best we can hope to do.

Claim 8.1 *Let k be the size of the smallest set cover for the instance (X, S). Then the greedy heuristic finds a set cover of size at most $k \ln n$.*

Proof: Let $Y_i \subseteq X$ be the set of elements that are still not covered after i sets have been chosen with the greedy heuristic. Clearly $Y_0 = X$. We claim that there must be a set $A \in S$ such that $|A \cap Y_i| \geq |Y_i|/k$. To see this, consider the sets in the optimal set cover of X. These sets cover Y_i, and there are only k of them, so one of these sets must cover at least a $1/k$ fraction of Y_i. Hence

$$|Y_{i+1}| \leq |Y_i| - |Y_i|/k = (1 - 1/k)|Y_i|,$$

and by induction,

$$|Y_i| \leq (1 - 1/k)^i |Y_0| = n(1 - 1/k)^i < n e^{-i/k},$$

where the last inequality uses the fact that $1 + x \leq e^x$ with equality iff $x = 0$. Hence when $i \geq k \ln n$ we have $|Y_i| < 1$, meaning there are no uncovered elements, and hence the greedy algorithm finds a set cover of size at most $k \ln n$. ∎

Exercise: Show that this bound is tight, up to constant factors. That is, give a family of examples where the set cover has size k and the greedy algorithm finds a cover of size $\Omega(k \ln n)$.