CSCI 3104, CU-Boulder **Prof. Aaron Clauset**
Lecture 8 **3 February 2014**

1 Greedy algorithms

In algorithm design, a *greedy algorithm* is one that breaks a problem down into a sequence of simple steps, each of which is chosen such that some measure of the "quality" of the intermediate solution increases after each step.

A classic example of a greedy approach is navigation in a k-dimensional Euclidian space. Let $\mathbf{y}$ denote the location of our destination, and let $\mathbf{x}$ denote our current position, with $\mathbf{x}, \mathbf{y} \in \mathbb{R}^k$. The distance remaining to our destination is then simply the Euclidian distance between $\mathbf{x}$ and $\mathbf{y}$:

$$L_2(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{k} (x_i - y_i)^2} \ , \tag{1}$$

and the greedy step to take at each moment in our journey is the one that minimizes $L_2(\mathbf{x}, \mathbf{y})$.[1]

Note that this approach is only *optimal*—that is, guarantees that we will always reach the target—when the space is smooth, with no impassable regions of space. It is not hard to construct examples in a 2-dimensional space containing impassable regions for which the greedy solution can fail. A trivial case is the following: consider traveling by car or foot from Boulder, CO to London, England. Ignoring the possibility of taking a ferry, there is no land path that connects North America to England, and thus there is no path.

Here is a more subtle case: consider traveling by car or foot from Miami, FL to Merida, Mexico (on the tip of the Yucatan Peninsula, across the Gulf of Mexico from Florida). In this case, a path does exist: travel counter-clockwise around the Gulf of Mexico, following the coast. However, a greedy approach cannot find this path because this path would first increase the distance to the destination, while the greedy approach always seeks to minimize the remaining distance. Thus, a greedy solution would take us south to the end of the Florida peninsula. Once there, we would find the part of the beach that is closest to Merida and then throw up our hands claiming that there is no new move that would not increase the remaining distance.

When a problem has these kinds of "local optima," from which no greedy move can further improve the quality of our position, we must use other means to solve it because a greedy approach can get stuck. (If there exists even a single input for which a greedy approach fails, then the greedy algorithm is not a correct algorithm.) But, many problems do have the particular kind of structure, if you can see it, that allows a correct greedy solution.

[1]We use L_2 (pronounced "L2") to denote the Euclidian distance because the general form of this distance measure is L_p, in which we take the pth power of each difference inside the sum and the pth root of the whole sum. In many problems, it is convenient to choose $p \neq 2$, which can produce a distance metric with slightly different mathematical properties.

1.1 Structure of a greedy algorithm

Greedy algorithms are possibly the most common type of algorithm because they are so simple to formulate. However, for a greedy algorithm to be correct (and fulfill the promise to never fail on any input), the problem must have the following mathematical properties:

1. Every solution to the problem can be assigned or translated in a numerical value or score. Let x be a candidate solution and let score(x) be the value assigned to that solution.
2. The "best" (optimal) solution has the highest value among all solutions (in the case of maximization; lowest, in the case of minimization); i.e., it is the global optimum, and this solution contains optimal solutions to all of its subproblems ("optimal substructure" property).
3. A solution can be constructed incrementally (and a partial solution assigned a score) without reference to future decisions, past decisions or the set of possible solutions to the problem ("greedy choice" property).
4. At each intermediate step in constructing or finding a solution, there are a set of options for which piece to add next.

A greedy algorithm *always chooses the incremental option that yields the largest improvement in the intermediate solution's score.*

As with any algorithm, once we have formulated our greedy approach, we must prove that it is correct, i.e., always finds the correct solution given any arbitrary input. Similarly, we must also analyze its time and space usage. These three parts (correctness, time usage, space usage) are what is typically meant by "analyzing" an algorithm

1.2 Huffman codes

A classic example of a greedy algorithm is Huffman encoding, a problem from compression.

In this area, we aim to take an existing "message," or rather a sequence of symbols, and translate it into a new message, composed with a different set of symbols, such that the new message is as small as possible while still containing all the information contained in the original one. You may be familiar with this idea from compressing files on your computer, e.g., making `.zip` or `.tgz` files. This form of compression is called *lossless* because the original message can be exactly reconstructed from the compressed one, with a little computation. If some information is lost, as in `.jpg` files, it is called *lossy compression.*

Here is the problem description for Huffman encoding. Let Σ, with $|\Sigma| = n$, denote the input alphabet, a set of distinct "symbols" that can be used to construct a string of symbols or message.

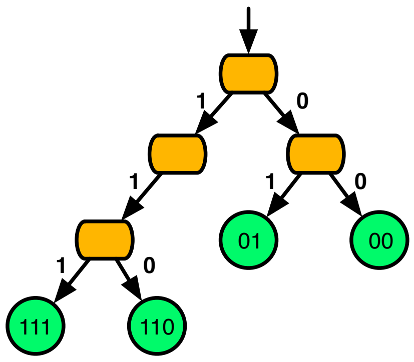
The input is a message or file x that is an arbitrary sequence of symbols drawn from Σ. The output must be an encoding (mapping) g of words from Σ to codewords in a binary alphabet Γ, such that the encoding is *prefix-free* and the encoded message minimal in size. In transforming the input to output, we aim to be as efficient as possible both in time (computation) and bits (encoded message size).

1.2.1 Binary code trees and prefix-free codes

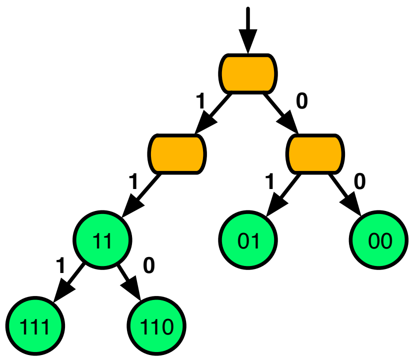
Before covering Huffman's algorithm, we must first define terms like binary codes, prefix-free codes, and message size. This subsection will cover the first two, and the next subsection the third.

A binary code is simply a set of strings (set of codewords) all drawn from the binary alphabet $\Sigma_{01} = \{0, 1\}$. A prefix-free code is a set of codewords such that no codeword is a prefix of any other. That is, if x is in the code, then there can exist no codeword $y = x + z$, where $+$ denotes concatenation. Otherwise, x is a prefix of y. If and only if a (binary) code is prefix-free, can it be represented by a (binary) tree in which codewords are only located at the bottom of the tree, at the leaves.[2,3]

To illustrate these ideas, below are two decoding trees, one that represents a prefix-free code, with codewords $\{111, 01, 110, 00\}$, and one that does not, which has codewords $\{11, 111, 01, 110, 00\}$. (Note that the second code is not prefix-free because 11 is a prefix of both 111 and 110.)



code $= \{111, 110, 01, 00\}$



code $= \{11, 111, 110, 01, 00\}$

[2]Note that this is different from the *trie* data structure, which explicitly permits words to be prefixes of each other. Tries are very handy data structures for storing character strings in a space-efficient manner that allows fast lookups based on partial (prefix) matchings.

[3]Also note that this kind of tree is very different from a search tree. In fact, the ordering of the leaves in an encoding is very unlikely to follow any particular pattern.

In the coding trees, codewords are denoted by green circles, with the corresponding codeword contained within and non-codewords are denoted by orange oblongs. To decode a string, we treat the tree like a finite-state automaton: each character we read moves us from one state (node) to another; if we read a 1, we move to the left child; otherwise, we move to the right child. When we reach a codeword node, we know what codeword we have just read, and we can return to the top of the tree to begin reading the next codeword. Clearly, any codeword that sits at an internal node in the tree is a prefix for all codewords in the subtree below it. We want to avoid prefix codes because they induce ambiguity in the decoding.

1.2.2 Cost of coding

The length of a codeword is given by the depth of its associated node. Thus, a good compression scheme is equivalent to a small tree, and the best compression is achieved by a minimal tree.[4] However, the total "cost" of an encoding scheme is not just a function of the size of the tree, but also a function of the frequency of the words we encode. Let f_i be the frequency in the message x of the ith codeword in the original message. A minimal code minimizes the function

$$\text{cost}(x) = \sum_{i=1}^{n} f_i \cdot \text{depth}(i) \ \ .$$

In 1948, Claude Shannon proved that the theoretical lower bound on the cost per word (in bits) using a binary encoding is given by the entropy $H = -\sum_{i=1}^{n} p_i \log_2 p_i$, where $p_i = f_i/n$. Notice that the definition of entropy is similar the cost function we wrote down, when $\text{depth}(i) = \log_2 f_i$. That is not an accident.

1.2.3 Huffman's algorithm

In 1952 David Huffman developed a greedy algorithm that produces an encoding that minimizes this function and gets as close to Shannon's bound as possible for a finite-sized string. The key idea of Huffman's algorithm is remarkably compact: merge the two least frequent "symbols" and recurse.

Huffman encoding begins by first tabulating the frequencies f_i of each word in x; this can be done quickly by constructing a histogram. We then create the n leaves of the coding tree. At the ith leaf, we store the ith word of the input alphabet Σ and its frequency f_i in the input message x.

Pseudocode for this procedure is straightforward, where we let f be an array of length n containing the given frequencies. The output will be an encoding tree with n leaves.

[4] "A" minimal tree, rather than "the" minimal tree, as there may be multiple distinct smallest trees for the same set of input symbols and frequencies.

```
Huffman(f) {
   initialize H              // H, a priority queue of integers, ordered by f
   for i = 1 to n {   insert(H,i)   }     // build queue
   for k=n+1 to 2n-1 {      // merge two least-frequent symbols and recurse
      i = deletemin(H), j = deletemin(H)
      create a node numbered k with children i,j
      f[k] = f[i] + f[j]
      insert(H,k)
   }
}
```

Let Γ denote the set of symbols we are currently working with. Initially $\Gamma = \Sigma$. At the kth step of the algorithm, we select the two words in Γ with smallest frequencies, f_i and f_j. We create a new word $n+k$ with frequency $f_{n+k} = f_i + f_j$, and make it the parent of i and j. We then update Γ by removing words i and j and adding word $n+k$. The algorithm halts when $|\Gamma| = 1$, i.e., when no pair of symbols remains, and returns to us the constructed tree.

Because the size of our intermediate alphabet Γ decreases by one at each step, the algorithm must terminate after exactly $n-1$ steps. Implementing the algorithm simply requires a data structure that allows us to efficiently find the two symbols in Γ with the smallest frequencies. This is typically done using a *priority queue* or *min heap* (Chapter 6). Each find, merge, and add cycle takes $O(\log n)$ time and there are n such operations; thus, the running time is $O(n \log n)$.[5] (How much space does it take?)

1.2.4 A small example

To illustrate how Huffman encoding works, consider the following set of symbols and their frequencies in a message to be encoded; for the encoding, the message itself doesn't matter, only the symbols and their frequencies.

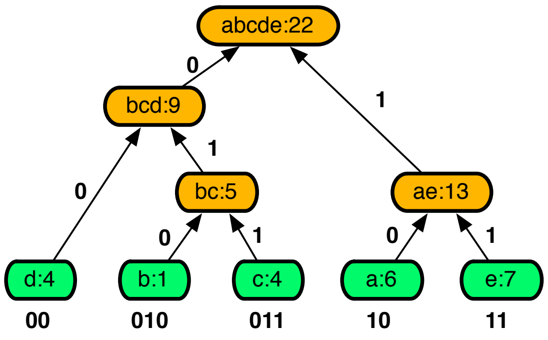
`a:6   b:1   c:4   d:4   e:7`

Huffman's algorithm proceeds by inserting each of these symbols into a priority queue data structure, in ascending order of their frequency. If two symbols have the same frequency, their order in the queue is arbitrary. Thus, the queue's initial contents are `(b:1)(c:4)(d:4)(a:6)(e:7)`.

[5] It is possible to run this sequence in $O(n)$ time using a pair of cross-liked queues; however, this version is more complicated to implement and assumes the word frequencies f_i have already been computed and sorted, which takes $O(n \log n)$ time.

We then repeatedly dequeue the first two elements i and j, with frequencies f_i and f_j, merge them into a new symbol whose frequency is $f_i + f_j$, which we insert back into the queue. (Remember that in a priority queue, the inserted new symbol will move up from the back of the queue until it finds its correct location.)

We repeat this process until the queue is empty, in which case the we have just dequeued the last two symbols and the newly merged symbol represents the entire set of symbols Σ and their number in the message. Applied to our example, we produce the following Huffman tree. Note that two symbols `c` and `d` have the same frequency; swapping their locations in the tree would produce a different optimal encoding. Similarly, we have arbitrarily labeled the left-child connection with 0 and the right-child connection with 1 to produce a set of codewords; making the symmetric choice would result in a different optimal encoding (in terms of the precise code words).



1.2.5 A cute example

Here is a cute "self-descriptive" Huffman example from Lee Sallows[6]

> This sentence contains three a's, three c's, two d's, twenty-six e's, five f's, three g's, eight h's, thirteen i's, two l's, sixteen n's, nine o's, six r's, twenty-seven s's, twenty-two t's, two u's, five v's, eight w's, four x's, five y's, and only one z.

To keep things simple, we will ignore capitalization, the spaces (44), apostrophes (19), commas (19), hyphens (3) and the one period in the sentence; instead, we will focus on encoding the letters alone. The frequencies of the 26 letters are the following.

[6]A. K. Dewdney. Computer recreations. *Scientific American*, October 1984. Other examples appeared a few years earlier in some of Douglas Hofstadter's columns. My credit goes to Jeff Erickson, who also produced the figure below.

A	C	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1

Below is the encoding tree produced by applying Huffman's rule to this histogram: after 19 merges, all 20 characters have been merged and the history of merges gives the encoding tree. (Suppose there is a tie as to which pair of characters to merge; what does this tell us about the uniqueness of the Huffman code?)

To read the encoding of a character, place a 0 on each left-branch and a 1 on each right-branch, then write the sequence of 1s and 0s backwards as you read up the tree. To decode an encoded letter, do the reverse: start at the root, and take the left-right path given by the encoding down the tree to arrive at the decoded character. For example, the encoding of the letter "a" is 110000. In our example, the encoded message is 661 bits long. Here is a table showing how we arrive at that number, with each input symbol, its frequency f_i in the input, the length of its encoding d_i and the total cost for encoding those symbols $f_i d_i$. The total message length is just $\sum_i f_i d_i$.

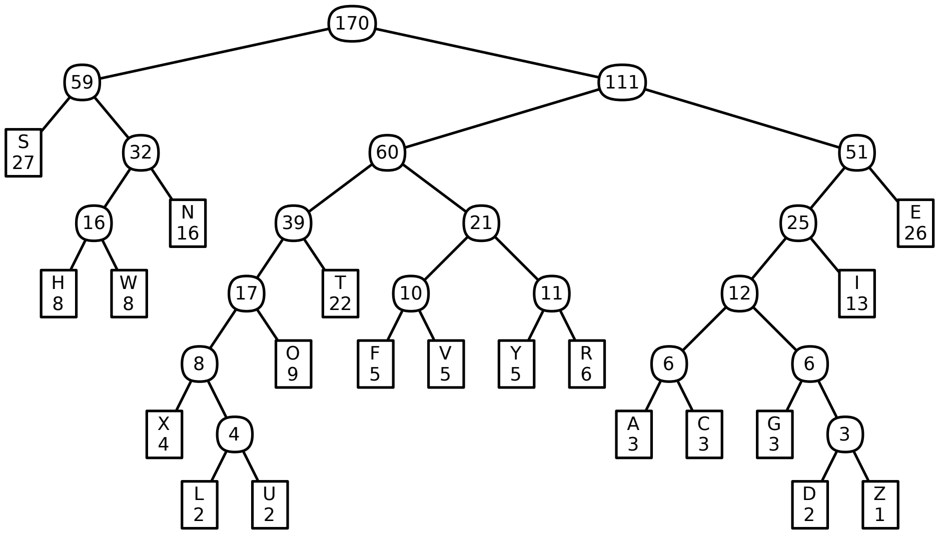
	A	C	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
f_i	3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1
d_i	6	6	7	3	5	6	4	4	7	3	5	5	2	4	7	5	4	6	5	7
$f_i d_i$	18	18	14	78	25	18	32	54	14	48	45	30	54	88	14	25	32	24	25	7

At home exercise: verify that this tree is correct.[7]

2 On your own

1. Read Chapter 16.2-3 (Elements of the greedy strategy and Huffman codes)

[7]This tree is not, in fact, correct. The correct tree produces an encoding that uses only 649 bits. Can you find the mistake?


170
59
111
S 27
32
60
51
16
N 16
39
21
25
E 26
H 8
W 8
17
T 22
10
11
12
I 13
8
O 9
F 5
V 5
Y 5
R 6
6
6
X 4
4
A 3
C 3
G 3
3
L 2
U 2
D 2
Z 1