Illinois Institute of Technology
Department of Computer Science

Lecture 12: October 8, 2014

CS 430 Introduction to Algorithms
Fall Semester, 2014

`"Greed is good. Greed works."` –Gordon Gekko (the 1987 movie *Wall Street*).

`"Greedy is pretty good. Greedy sometimes works."` –Edward Reingold (CS 430, Fall, 2014).

The Greedy Heuristic

Greedy algorithms make decisions according to a greedy heuristic which chooses the best local decision possible without paying attention to the long-term consequences. For example, a greedy hare competing with a turtle, might run as fast as he can for as long as he can, and then have to rest for a long time because he is tired. The non-greedy turtle, on the other hand, might conserve his energy and be able to beat the hare by not requiring a rest.

Greedy algorithms are appropriate for many types of problems, and we will look at some examples where the greedy algorithm works well.

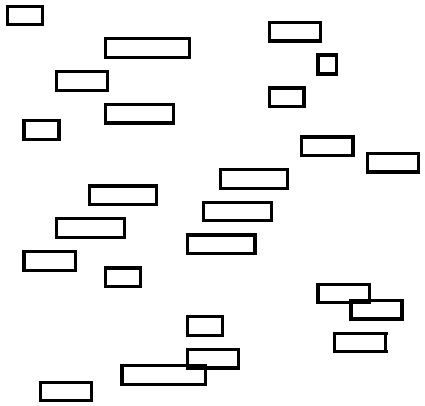
Activity Selection

Consider a simplified version of job scheduling:

You are given a list of jobs $J_1, J_2, J_3, \ldots J_n$ to run on a single processor; each job has a start time s_i when it must start and finish time f_i at which point it must conclude. However, the processor can only run one job at any given moment. Your task is to schedule the maximum number of jobs to run on the processor.

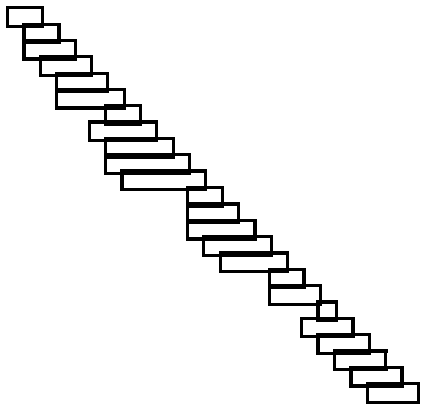
Clearly this formulation of job scheduling is unrealistic because it ignores important constraints such as the fact that some jobs are more important than others and that some jobs absolutely must be run for the system to remain operational, etc. It is precisely this simplistic formulation which allows the greedy heuristic to work well.

Consider the following list of jobs:

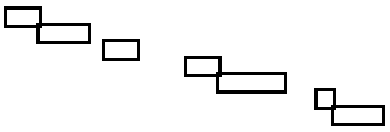


There are many ways to schedule these jobs. In the exhaustive scheduling method, we could just examine every subset of these jobs and see which subset has the largest cardinality (i.e. number of jobs). There are 2^n ways to make a subset of a set of size n, so our exhaustive algorithm will require exponential time.

If we allow for some preprocessing, we could significantly improve this algorithm using the greedy heuristic. Specifically, consider first sorting the jobs in ascending order of finish time:



Now, we may start at the top of our sorted list and greedily include the first job in our schedule. The second and third jobs conflict with our current job selection, so we skip over them and add the fourth job to our schedule. In this way we can go through the entire list of jobs, greedily adding jobs that fit our schedule to get the following list of jobs:



This method is called the `GREEDY ACTIVITY SELECTOR`, where "activity" refers to a job, in the current context. The code for `GREEDY ACTIVITY SELECTOR` may be found in CLRS (page 421). We will now analyze the running time of this algorithm.

`GREEDY ACTIVITY SELECTOR` visits each item to be scheduled once, therefore the selection part requires time proportional to n. Remember that we first sorted the list, which required $O(n \log n)$ time, so the entire algorithm has a complexity of $O(n + n \log n)$ which is $O(n \lg n)$. This is much better than 2^n for the

exhaustive algorithm. However, recall that the exhaustive algorithm guaranteed us an optimal scheduling; does this algorithm give us an optimal solution? In other words, does making locally optimal ("greedy") decisions give a globally optimal solution in this case?

The answer is "yes" and we prove the optimality of our greedy algorithm by comparing the set selected with the greedy algorithm (called A) with a true optimal set (called B). If A is not optimal, there must be a place where the greedy algorithm selected a non-globally-optimal choice. If a non-optimal choice was made, there must be a first non-optimal choice. Let's see where that first non-optimal choice could have been.

Proof of the Optimality of the Greedy Algorithm

Let $A = \{a_{x_0}, a_{x_1}, a_{x_2}, a_{x_3}, \cdots\}$ be the greedily chosen set of jobs. *Assume* that after picking $j-1$ jobs correctly, the greedy algorithm makes a mistake. It picks an activity a_{x_j} that could not possibly lead to an optimal solution. Let us say that a correct choice of jobs was b_{x_j} that would lead to an optimal solution $B = \{a_{x_0}, a_{x_1}, a_{x_2}, a_{x_3}, \cdots, a_{x_{j-1}}, b_{x_j}, b_{x_{j+1}}, \cdots\}$.

Clearly, a_{x_j} must end before b_{x_j} or else our greedy selection would have picked b_{x_j}. Thus, switching a_{x_j} and b_{x_j} in the optimal solution B (to get a selection B') must still give a valid activity selection (think about this: remember that a_{x_j} was an acceptable choice for jobs after $j-1$ iterations). However, B' contains the same number of elements as B and is, hence, optimal. Thus, the greedy choice does indeed provide the possibility for an optimal solution, contrary to our original assumption. Since each choice of jobs by the greedy strategy allows for an optimal solution, and we continue choosing jobs until exhaustion, the greedy activity selection must be optimal overall.

Huffman Codes

We continue our discussion of greedy algorithms with the examination of Huffman codes.

Consider the problem of encoding the alphabet [a,b,c,...,z] in binary. We can use either 0's and 1's (or dots and dashes like in Morse Code). Each letter has to have a unique encoding.

If we represent all the letters in binary, we will need at least $\lg(26) \approx 4.70$ bits. Since the number of bits has to be integral, we will use 5 bits to represent each letter. We can generate the following encoding (Notice that 11010 through 11111 are not assigned a letter and are therefore invalid encodings in this context):

00000	a	00111	h	01110	o	10101	v
00001	b	01000	i	01111	p	10110	w
00010	c	01001	j	10000	q	10111	x
00011	d	01010	k	10001	r	11000	y
00100	e	01011	l	10010	s	11001	z
00101	f	01100	m	10011	t		
00110	g	01101	n	10100	u		

If we had a simple message such as "One if by land ...", How long would the encoding be? For this encoding, exactly five characters are used for every letter, so since the above phrase of 11 characters (not including spaces and punctuation) would take 55 bits to encode.

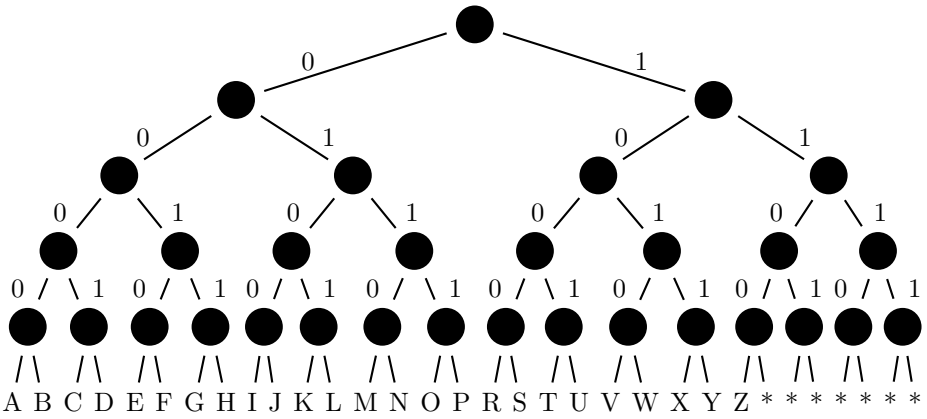
The fact is, however, that in English, for example, some letters are used more often than others. "E" is used quite often while "Z" and "Q" are used rarely (some might say we could even do without them altogether). Suppose we want to design a code that will take these frequencies into account. Specifically, we would encode letter "E" with very few bits, and allow the letter "Q" to be encoded with many bits. The problem now would be that it is hard to tell when one letter ends and another letter begins. Consider a partial listing of Morse Code, for example:

·	e	··	i	—	t	——	m
·—	a	···	s	—·	n	··—	u

If I gave the encoded string "···—" you would not know whether it represents "eeet" or "eit" or "iet" or "st" or "ia" or "eea", etc.

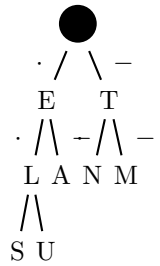
For this reason, we desire our encodings to be prefix-free. A *prefix-free code* (also known as a *prefix code*) is a code in which no one encoding is a prefix of any other encoding. For example, Morse Code is **not** prefix-free because the encoding for "e" is a prefix of the encoding for "i" which is also a prefix of the encoding for "s".

Our first encoding of the English letters, on the other hand, **is** prefix-free. No one encoding is a prefix of the other. This prefix-free property allows us to arrange the encodings into a binary tree as follows:



The "*"'s denote unused encodings. Notice that each letter is at a leaf, which implies that the code is a prefix-free code. If someone gave us the encoded message "00110100", we could merely follow the strings down the binary tree; whenever we reach a leaf, we can output the letter at that leaf and then start at the root of the tree again to decode the next letter. In this case, we decode to "HI" (you might want to actually try this decoding at home...don't worry, it's safe).

On the other hand, consider a partial tree for Morse Code:



Here the internal nodes are used as letters, which implies that Morse Code is **not** prefix-free.

Generating Huffman Codes

A Huffman code is a prefix-free code that minimizes the average (or expected) length of letter encodings. Specifically, if letter l_i has a probability p_i of occurring and requires $\text{depth}(l_i)$ encoding bits, then the expected length of an encoded letter is:

$$E(\text{letter}) = \sum_{i \in \text{alphabet}} p_i \cdot \text{depth}(l_i)$$

Multiplying the expected length by the number of symbols in the data gives us a quantity known as the weighted path length (WPL) of the corresponding encoding tree:

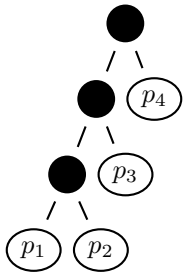
$$WPL = \sum_{i \in \text{alphabet}} \text{frequency}(i) \cdot \text{depth}(l_i)$$

In our first letter encoding, if the i'th letter has probability p_i of occurring, then the expected encoding length is $E(\text{letter}) = \sum_{i \in \text{alphabet}} p_i \cdot 5$ since each letter is at depth 5. We can manipulate this expression to get that:

$$\begin{aligned} E(\text{letter}) &= \sum_{i \in \text{alphabet}} p_i \cdot 5 \\ &= 5 \sum_{i \in \text{alphabet}} p_i \\ &= 5 \cdot 1 = 5 \end{aligned}$$

since the sum of all the probabilities of all the letters must always equal 1. This expected length is somewhat obvious since each encoding has length 5.

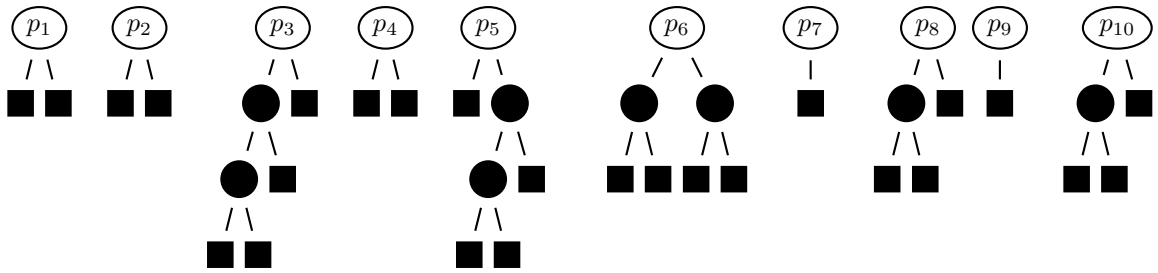
For another example, consider the following tree with probabilities of p_4, p_3, p_2 and p_1:



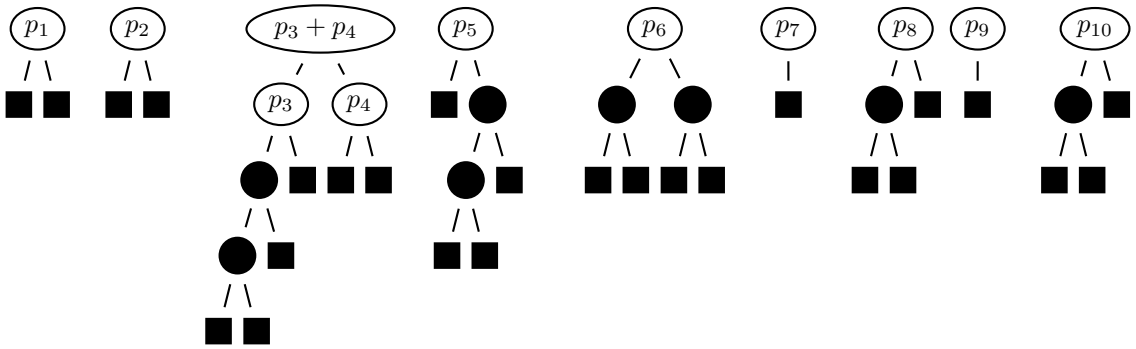
This code has an expected encoding length of $3p_1 + 3p_2 + 2p_3 + p_4$.

To generate our Huffman code we will start off by giving each letter its own one-vertex tree, whose value will be the probability of that letter. We will then employ a greedy heuristic to merge some of these trees together. Specifically, we will merge the two trees with the *least* values. The value of this merged tree will be the sum of the values of the two component trees.

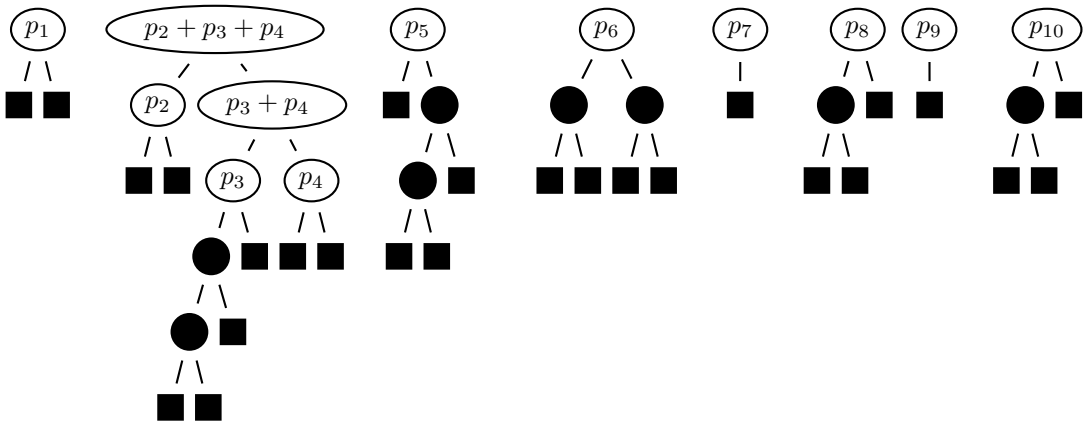
For example, suppose we get to a stage of our computation where we have the following trees (the value of each tree is printed in the root node):



Suppose that p_3 and p_4 are the lowest values. Then we would merge those two trees to get the following new collection of trees:



Again we would combine the two lowest values present. For example, suppose that $p_3 + p_4$ and p_2 are now the two lowest values. We would merge them to get the following set of trees:



and so forth until we have only one tree representing our Huffman code. The details of this algorithm are also explained well in CLRS, section 16.3.

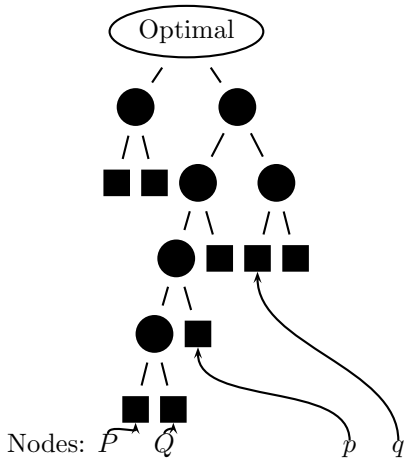
Optimality of Huffman Codes

We must wonder whether this greedy strategy really produces an optimal code. In this case, an optimal tree has the lowest expected encoding length. We will prove the optimality of this strategy in the same way that

we proved the optimality of our greedy activity selector. Our proof in these lecture notes is sketchy because a rigorous proof requires tedious attention to some important details. You should examine CLRS for a more complete proof.

Suppose we have been constructing trees, and have gotten to a certain set of trees that can be appropriately merged to an optimal tree, but that we make the wrong (greedy) choice for which trees to merge. In other words, the greedy merging leaves us with a set of trees that cannot be merged to make an optimal tree. By appropriately transforming our alphabet, we may assume that the greedy algorithm's mistake is made on the very first step.

In this case, the greedy algorithm must have made two symbols siblings, whereas the optimal algorithm separated them:



Call the lowest two nodes (i.e. greatest depth) in the optimal tree P and Q and denote their length L. Call the lowest frequency nodes in the optimal tree p and q and call their depth l.

We now claim that exchanging $P \longleftrightarrow p$ and $Q \longleftrightarrow q$ does not increase the expected symbol length, implying that the greedy choice can, in fact, lead to an optimal encoding.

To prove this claim, note that exchanging $P \longleftrightarrow p$ will increase the expected length by $lP - Lp$, where P and p refer to the frequencies of the respective nodes. However, since $l \leq L$ and $p \leq P$ (by the greedy heuristic), $lP - Lp \leq 0$ so that the expected length cannot increase. Moreover, it must be that $p = P$ since P belongs to an optimal tree. The same argument applies to q and leads to the result that $q = Q$.

As a side note, we may ask how to implement this algorithm; for instance, what data structure should we use? The answer is that a heap or, more generally, a priority queue (first-in, highest-priority out) would be best suited for this task. We would then need n insertions for each of the symbols in the initialization stage, and $n-1$ `delete`, `delete`, `insert` combinations for each of the tree mergings, resulting in $2n-1$ operations in total. Each operation requires $\Theta(\log n)$ time giving an overall running time of $O(n \lg n)$.

Greedy algorithms tend to be simple and efficient, as the locally optimal solution to a problem is usually easy to find. Unfortunately, greedy algorithms do *not* always yield optimal solutions.

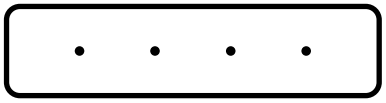
Greedy algorithms and suboptimal solutions

Suppose you are given a set of points on the plane. Consider the problem of connecting each point to exactly one other point in such a way that the cost, the sum of the lengths of the edges, is minimized.

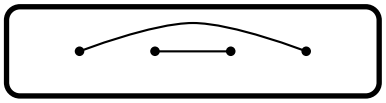
A brute force approach to this problem is to generate each possible matching of points and to find the corresponding costs. The matching with the minimum cost is thus the solution to the problem. While this approach will inarguably produce correct results, it does require exponential time. This encourages us to search for a more efficient algorithm.

One simple algorithm that comes to mind is a greedy algorithm. Find the two nearest points that are not yet connected (to other points) and connect them. Repeat this process exhaustively. In the case of a tie, select a pair of points arbitrarily. How efficient is this algorithm? If there are n points, it will take time $\binom{n}{2} + \binom{n-2}{2} + \binom{n-4}{2} + \cdots = \Theta(n^3)$ to iteratively look for two nearest points.

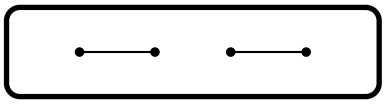
Before declaring success, we must ask if our algorithm is correct: does it, in fact, give optimal results? Unfortunately it does not. Consider four points equally spaced along a line:



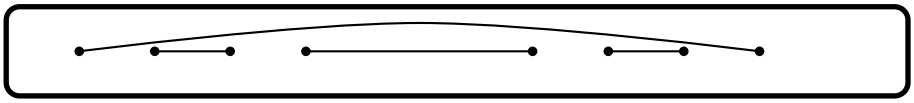
In the first iteration, there are three optimal pairs of points, so the algorithm chooses a pair arbitrarily. Say it chooses the two points in the middle and is then forced to connect the two remaining points: [1]



This is clearly not the optimal solution, however. The optimal solution connects the two leftmost points and the two rightmost points:



In fact, we can exploit the above problem with the greedy algorithm by putting two instances of the above problem side by side:



Again, the optimal solution does much better:

[1] This choice of the middle points first can be forced by separating the points by distances 1, $1 - \epsilon$, and 1, where ϵ is some small positive number.

We can repeat this process to get examples where the greedy performance gets worse and worse. Specifically, consider repeating this process until we have $N = 2^n$ points. The minimum cost of the optimal algorithm is:

$$OPT_n = 2^{n-1}, \text{ where } OPT_1 = 1$$

Compare this to the solution given by the greedy algorithm. Let L_n be the length from the leftmost point to the rightmost point if there are 2^n points. This leads to the recurrence:

$$\begin{aligned} GREEDY_n &= 2GREEDY_{n-1} + L_n - 2L_{n-1} + L_{n-1} \\ &= 2GREEDY_{n-1} + L_n - L_{n-1} \\ &= 2GREEDY_{n-1} + 3^{n-1} - 3^{n-2} \end{aligned}$$

The recurrence is annihilated by $(\mathbf{E} - 2)(\mathbf{E} - 3)$, leading to the solution:

$$GREEDY_n = 2 \cdot 3^{n-1} - 2^{n-1}$$

The error in the greedy algorithm relative to the optimal solution is therefore:

$$\begin{aligned} \frac{GREEDY_n}{OPT_n} &= \frac{2 \cdot 3^{n-1} - 2^{n-1}}{2^{n-1}} \\ &= 2 \cdot \left(\frac{3}{2}\right)^{n-1} - 1 \\ &= \frac{4}{3}\left(\frac{3}{2}\right)^{n} - 1 \\ &= \frac{4}{3}\left(\frac{3}{2}\right)^{\lg N} n - 1 \\ &= \frac{4}{3}N^{\lg \frac{3}{2}} - 1 \\ &\approx O(N^{0.5849625\ldots}) \end{aligned}$$

Not only is the solution given by the greedy algorithm not optimal, its relative error grows with the size of the problem! Clearly, the greedy approach is not efficient, in the worst case, for this problem.

Another example where the greedy algorithm fails is the problem of "nicely" printing a binary tree as narrowly as possible, that we discussed in the first lecture. We noticed that, sometimes, the narrowest nice printing of the tree involved a wider-than-necessary nice printing of the subtrees.