Algorithm Design and Analysis 3/25/2008

Lecture 28: Greedy Algorithms

Instructor: Mike Kowalczyk

1 Greedy Algorithms

A greedy algorithm is one that makes choices that look optimal locally, without considering the entire structure of the problem. It does whatever looks good at the moment. They tend to be easy to invent. Also, they tend to have a fast runtime. We will also see a standard method for proving correctness of greedy algorithms.

1.1 A scheduling problem

Imagine there's one resource (a lecture hall), and a lot of different activities competing for it. We want to schedule the max possible number of activities in this room. Every activity has a start and end time. For example, we might have activities at (start, end] time intervals $[1,4)$, $[2,7)$, $[3,4)$, $[5,11)$, $[6,8)$. What's a maximum number of activies included, assuming there cannot be any overlap between different activities? We can use $[1,4)$ and $[5,11)$, and in this case, this is the best we can hope for.

Imgine lots of activies (n of them). We want a greedy algorithm (select what "looks best" at the moment to construct an optimal solution). One suggestion: Take the one that starts earliest and ends earliest. This seems to work good. What if no such activity exists? One possibility is select the activity with the shortest time first, but this might cause problems with, say, if we have activies like $[1,7)$, $[6,8)$, and $[7,10)$. The simplest way to attack this problem is to sort by finish time, and schedule the lowest possible finish time first.

We programmed the algorithm together in Java (see source).

Can we argue correctness? Yes. The general technique is to suppose that some other choice (i.e. other than the one our algorithm actually makes) leads to an optimal solution, and then we argue that our choice (the choices our algorithm actually makes) must also lead to an optimal solution. In the case of our activity scheduling problem, the first activity, a, selected is one with earliest finish time. Suppose that there was some other choice for the first activity, b, which is used in optimal schedule B (which has the maximum number of activities). Then since $\text{finishTime}[a] \leq \text{finishTime}[b]$ we can replace b with a in schedule B and still have an optimal schedule. Therefore, we can choose an activity with earliest finish time for the first activity, and we know there is at least one choice for the later activites which is optimal. Note that this same argument applies to any iteration of the algorithm, because at any step a smaller instance of the same scheduling problem is induced by considering all activities with start time greater than or equal to the finish time for the latest-selected activity.

How about runtime? In theta notation, how many operations (worst case) does this program take to run (on n activites)? The runtime is easy to calculate and is $\Theta(n^2 + n) = \Theta(n^2)$. But if we put in MergeSort, then our runtime is $\Theta(n + n \lg n) = \Theta(n \lg n)$. If we assume that the input was already sorted by finishTime, then we can remove the sort and the runtime becomes $\Theta(n)$.