CSC373S Lecture 6

- To conclude our current discussion of greedy algorithms, lets summarize the greedy paradigm:

  1. Fixed order greedy ("greedy-like") template:
     Use a "local ordering rule" to order the input items so that they now appear in the order $I_1, \ldots, I_n$
     For $i = 1..n$
     Make a greedy (perhaps non greedy) irrevocable decision about I_j
     End For
     Examples: The EFT greedy alg for interval selection, the EST greedy alg for interval colouring, Kruskals MST alg, the online and LPT greedy algs for makespan on m identical machines.
  2. Adaptive order greedy (greedy-like) template:
     While there are input items remaining
     Use a local ordering rule (based on what has been done so far) to choose next input item I
     Make a greedy (perhaps non greedy) irrevocable decision about I_j
     End While

     Examples: The greedy alg for interval selection that takes the next interval having the least number of conflicts with intervals not yet eliminated, Prim's MST alg, Dijkstra's least cost paths alg, Huffman prefix codes.

  I have not defined what constitutes a "local ordering". One definition is that it is an ordering defined by a function f mapping input items to real numbers and then use the ordering induced by sorting the numbers $\{f(I_j) : 1 \leq j \leq n\}$. And what do we mean by a "greedy decision". I mean informally, do what seems to be best for the input item being considered ignoring any possible consequences for the future; that is, "live for today". A non-greedy decision (and hence a "greedy-like" algorithm) has the freedom to make non greedy decision so as to plan for possible future (i.e.not yet considered) input items. I briefly indicated that one can improve upon the approximation of the online greedy algorithm (and still be online in the sense of not being able to sort the inputs) by leaving some room on a lightly loaded machine to accommodate a job having a large load. There are many ways to generalize what we might consider to be a greedy algorithm but I claim that all of the greedy algorithms that we have studied thus far fit the above templates.

- We will now take a deviation from the order of topics in the text to consider dynamic programming DP and then follow this by divide and conquer, whereas the text first does divide and conquer. I want to first do DP as it is easy to motivate DP algorithms by considering two problems (already considered) where we cannot extend the greedy solutions to apply to a more general problem.

- We start our discussion of DP algorithms by considering the weighted interval selection problem. In this problem, intervals now have weights or values so that an interval $J(i) = (s_i, f_i, w_i)$ where w_i is the weight of the i^{th} interval. (Previously

we studied the unweighted interval selection problem for which EFT is an optimal algorithm.) As the Jepson notes show, if we use the EFT algorithm for the weighted problem, the solution can be arbitrarily bad. Similarly, we can try any of the other suggested greedy algs for the unweighted case and they can also be shown to be arbitrarily bad. This raises the question as to whether or not there exists any greedy algorithm that is optimal or even attains a constant approximation for the weighted interval selection problem.

To answer such a question one has to have a precise definition for what will be considered a greedy algorithm and I claim that with regard to the given greedy (or greedy-like) templates, that one can prove that there does not exist a constant approximation greedy algorithm for weighted interval selection. There are some extensions of the given templates that may or may not be considered greedy but do obtain a constant approximation. And if one extends the concept sufficiently then one can obtain optimality but I believe that such extensions are not what we intuitively tend to consider to be greedy algorithms.

- Returning to the problem itself (weighted interval selection) we now consider how dynamic programming provides an optimal solution. See the dynamic programming notes, slides 2-4. Motivation: look at an optimal solution OPT. Look at the rightmost interval in OPT, and say it is $J_j = (s_j, f_j)$. Then when we remove J_j, what remains must be an optimal solution amongst intervals that end before s_j.

  Aside: I have been thinking of intervals as closed intervals $[s_j, f_j]$ whereas the notes now views then as open intervals (s_j, f_j) meaning that the intervals (s, t) and (t, f) do not intersect. Everything we have been doing can be made to work for either view (ie open or closed intervals) but to be consistent with the notes lets now view intervals as being open.

  Once we recognize this "subproblem optimality", it is not difficult to see how to obtain a recursively defined optimal algorithm. Namely, we again sort the intervals so that $f_1 \leq f_2 \ldots \leq f_n$ and then compute $p(j) = \max\{i : f_i \leq s_j\}$ for all $j : 1 \leq j \leq n$. If we were considering closed intervals then we would define $p(j) = \max\{i : f_i < s_j\}$.

  Having computed $p(j)$ for $j = 1, \ldots, n$ (which can be done in time $O(n \log n)$), we then have the following DP algorithm which will optimally compute the value of an optimal solution.

$$\begin{aligned} OPT(j) &= 0 \text{ if } \quad j = 0 \\ &= \max\{OPT(j-1), w_j + OPT(p(j)\} \quad \text{if } j > 0. \end{aligned}$$

  The first option $OPT(j-1)$ corresponds to the situation where the j^{th} interval J_j is not needed in an optimal solution and the second option corresponds to the case where we do want to include J_j is an optimal solution. It can be the case that there are optimal solutions with and without J_j and we can then use either solution. (The

final question on the assignment asks you to count the number of different optimal solutions for a given instance of the problem.)

Note: It is standard to think about optimal DP algorithms (for an optimization problem) by first computing the value of an optimal solution. We will see it is easy to then convert this to an algorithm that computes an optimal solution. That is, in determining the optimal value using dynamic programming we are implicitly showing how to compute an optimal solution corresponding to the optimal value.

- The way I like to think about greedy algorithms is to first try to define a "semantic" array that indicates what we are trying to compute. In the case of weighted interval selection, we are trying to compute $OPT'(j)$= the optimal value that can be obtained by a non conflicting subset of the intervals $J_1, \ldots J_j$. I am using OPT' to distinguish it from the recursively defined array OPT. Formally, one needs to prove that $OPT'(j) = OPT(j)$ for all j. We usually do this somewhat informally (often waiving our hands) but we really should have a proof that the two arrays are equivalent. Of course, the array OPT' is defined in such a way that it 1) will be easy to obtain the desired answer from one or more entries of the array and 2) every entry of the array can be easily computed from "previous" entries of the array. That is, when we define the array OPT' we usually have some recursively defined way of computing it (i.e. OPT)

- The slides show that a naive implementation of OPT will be very inefficient (exponential time) but that the use of *memoization* provides an efficient recursive implementation. And there is a very efficient iterative implementation. One usually uses the iterative implementation but usually the conception of the algorithm is in terms of the recursive definition.

- Finally, we address the question as to how to compute an optimal solution and not just the value of an optimal solution. The slides give a recursive method for constructing an optimal solution once the optimal value has be determined. Another method (which may not be as efficient but is conceptually simpler) is to construct the solution while computing the optimal value. Namely, when one chooses between not including (resp. including) J_j in computing $OPT(j)$, we can add a line of code $SOL(j) := SOL(j-1)$ (resp. $SOL(j) := SOL(j-1) \cup \{J_j\}$) according to which choice afford the higher value.