CS 6783 (Applied Algorithms) – Lecture 3

Antonina Kolokolova*

January 13, 2012

1 Greedy Algorithms

One algorithm design method on which many fast algorithms and heuristics are based is the greedy method. This method relies on the ability to achieve the globally optimal solution by making a sequence of locally optimal choices. In the greedy method, once the choice is made it is never reverted (otherwise it is a case of a more general backtracking method).

Let's consider an example (called in KT the "interval scheduling" problem).

1.1 Activity selection

Consider the following scenario. A university has a large lecture hall, and would like to schedule as many different activities in that hall as possible. It is given a list of possible activities out of which it is making its selection, where an activity is defined to be a pair (s, f) of nonnegative integers such that $s < f$; the intuition is that s is the starting time of the activity and f is the finishing time of the activity. We will be given a sequence of activities, and we wish to schedule as many of them as possible (in one room) so that no two scheduled activities overlap. If (s, f) and (s', f') are activities, we say they *do not overlap* if $f \leq s'$ or $f' \leq s$.

More formally, the input is a sequence of n activities, $(s_1, f_1), (s_2, f_2), \cdots, (s_n, f_n)$. A schedule is defined to be a set $A \subseteq \{1, 2, \cdots, n\}$ such that for all $i, j \in A$, if $i \neq j$ then (s_i, f_i) does not overlap (s_j, f_j). The goal is to find a schedule A such that $|A|$ (the size of A) is as big as possible. (Since we are only interested in how many activities are in A, we are in effect treating each activity as if it yields unit profit. More complicated versions of this problem allow different activities to yield different profits.)

Our greedy algorithm will work as follows. First, we will sort the activities according to nondecreasing finish times. Then we will go through the activities, one at a time, scheduling each activity if possible. Before formally stating the algorithm, we give the following exercise.

Exercise: Prove that if, instead of sorting by nondecreasing finish times, we sort by nondecreasing start times, then the algorithm would not work.
Prove that if, instead of sorting by nondecreasing finish times, we sort by nondecreasing job size (that is, $f - s$), then the algorithm would not work.

We now give code for the algorithm. We will use a variable e to keep track of the last finish time of an

*This set of notes is based on the course notes of U. of Toronto CS 364 as taught by Stephen Cook

activity added to A, where $e = 0$ if A is empty. That is, since the jobs are sorted according to finish time, e is the earliest start time at which we can add an activity to A.

Greedy:
Sort the activities so that: $f_1 \leq f_2 \leq \ldots \leq f_n$
$A \leftarrow \emptyset$
$e \leftarrow 0$
for $i : 1..n$
    if $s_i \geq e$ then
        $A \leftarrow A \cup \{i\}$
        $e \leftarrow f_i$
    end if
end for

Analyzing the running time of the algorithm, we see that the $n \log n$ time for the sort dominates the linear time for the loop, so the total time is $O(n \log n)$.

A Greedy algorithm often begins with sorting the input data in some way. The algorithm then builds up a solution to the problem, one stage at a time. At each stage, we have a partial solution to the original problem – don't think of these as solutions to subproblems (although sometimes they are). At each stage we make some decision, usually to include or exclude some particular element from our solution; we never backtrack or change our mind. It is usually not hard to see that the algorithm eventually halts with some solution to the problem. It is also usually not hard to argue about the running time of the algorithm, and when it is hard to argue about the running time it is because of issues involved in the data structures used rather than with anything involving the greedy nature of the algorithm. The key issue is whether or not the algorithm finds an *optimal* solution, that is, a solution that minimizes or maximizes whatever quantity is supposed to be minimized or maximized. We say a greedy algorithm is optimal if it is guaranteed to find an optimal solution for every input.

Most greedy algorithms are not opitmal! The method we use to show that a greedy algorithm is optimal (when it is) often proceeds as follows. At each stage i, we define our partial solution to be *promising* if it can be extended to an optimal solution by using elements that haven't been considered yet by the algorithm; that is, a partial solution is promising after stage i if there exists an optimal solution that is consistent with all the decisions made through stage i by our partial solution. We prove the algorithm is optimal by fixing the input problem, and proving by induction on $i \geq 0$ that after stage i is performed, the partial solution obtained is promising. The base case of $i = 0$ is usually completely trivial: the partial solution after stage 0 is what we start with, which is usually the empty partial solution, which of course can be extended to an optimal solution. The hard part is always the induction step, which we prove as follows. Say that stage $i + 1$ occurs, and that the partial solution after stage i is S_i and that the partial solution after stage $i + 1$ is S_{i+1}, and we know that there is an optimal solution S_{opt} that extends S_i ; we want to prove that there is an optimal solution S'_{opt} that extends S_{i+1} . S_{i+1} extends S_i by making only one decision; if S_{opt} makes the same decision, then it also extends S_{i+1}, and we can just let $S'_{opt} = S_{opt}$ and we are done. The hard part of the induction step is if S_{opt} does not extend S_{i+1}. In this case, we have to show either that S_{opt} could not have been optimal (implying that this case cannot happen), or we show how to change some parts of S_{opt} to create a solution S'_{opt} such that

- S'_{opt} extends S_{i+1}, and
- S'_{opt} has value (cost, profit, or whatever it is we're measuring) at least as good as S_{opt}, so the fact that S_{opt} is optimal implies that S'_{opt} is optimal.

For most greedy algorithms, when it ends, it has constructed a solution that cannot be extended to any

solution other than itself. Therefore, if we have proven the above, we know that the solution constructed must be optimal.

Let us now use this method to prove the algorithm above for the activity scheduling problem does produce an optimal solution.

Theorem 1 *This Greedy algorithm outputs an optimal (that is, largest possible) schedule.*

Let A_i and e_i be the values of A and e after the body of the 'for' loop has been executed i times. It is easy to prove by induction on i that for all i, $0 \leq i \leq n$:
(*) $A_i \subseteq \{1, 2, \cdots, i\}$ and
(**) $e_i = \max\{f_j \mid j \in A_i\}$ (where $\max \emptyset = 0$).

The Theorem clearly follows from the following Lemma.

Lemma 1 *For $0 \leq i \leq n$, A_i is* promising *after stage i, that is, there exists an optimal schedule A_{opt} such that $A_i \subseteq A_{opt} \subseteq A_i \cup \{i+1, \cdots, n\}$.*

Proof of Lemma: Clearly A_0 is promising after stage 0.

So let $0 \leq i < n$ and assume that A_i is promising after stage i. We want to show A_{i+1} is promising after stage $i+1$. Let A_{opt} be an optimal schedule such that
$A_i \subseteq A_{opt} \subseteq A_i \cup \{i+1, \cdots, n\}$. Clearly (*) and (**), together with the fact that the finish times are in sorted order, imply that $e_i \leq f_{i+1}$.

Case 1: $s_{i+1} < e_i$, so $A_{i+1} = A_i$.
Then, since e_i is the finish time of an activity in A_i and $e_i \leq f_{i+1}$, we see that activity $i+1$ overlaps an activity in A_i, so $i+1 \notin A_{opt}$. So $A_{i+1} \subseteq A_{opt} \subseteq A_{i+1} \cup \{i+2, \cdots, n\}$.

Case 2: $s_{i+1} \geq e_i$, so $A_{i+1} = A_i \cup \{i+1\}$.

Subcase 2A: $i+1 \in A_{opt}$.
Then $A_{i+1} \subseteq A_{opt} \subseteq A_{i+1} \cup \{i+2, \cdots, n\}$.

Subcase 2B: $i+1 \notin A_{opt}$.
Since $s_{i+1} \geq e_i$, (**) implies that activity $i+1$ does not overlap any activity in A_i. A_{opt} cannot equal A_i, for then $A_{opt} \cup \{i+1\}$ would be a larger schedule than A_{opt}. So let $u \geq i+2$ be the activity in $A_{opt} - A_i$ with the smallest finish time. Consider the activities in $A_{opt} - A_i$ other than u: since these all have start times $\geq f_u$, and since $f_u \geq f_{i+1}$, we see that none of these activities overlap activity $i+1$. Let $A'_{opt} = (A_{opt} - \{u\}) \cup \{i+1\}$. A'_{opt} is a schedule, and since it has the same size as A_{opt}, it is an optimal schedule. We also clearly have $A_{i+1} \subseteq A'_{opt} \subseteq A_{i+1} \cup \{i+2, \cdots, n\}$. □

Exercise 1 *Generalize this to work with multiple rooms in which to schedule activities. In particular, generalize this algorithm so that it would produce a way to schedule all activities using the minimal number of (identical) rooms.*

1.2 A Greedy Algorithm for Scheduling Jobs with Deadlines and Profits

The setting is that we have n jobs, each of which takes unit time, and a processor on which we would like to schedule them in as profitable a manner as possible. Each job has a profit associated with it, as well as a deadline; if the job is not scheduled by the deadline, then we don't get the profit. Because each job takes the same amount of time, we will think of a Schedule S as consisting of a sequence of job "slots" $1, 2, 3, \ldots$

where $S(t)$ is the job scheduled in slot t.
(If one wishes, one can think of a job scheduled in slot t as beginning at time $t-1$ and ending at time t, but this is not really necessary.)

More formally, the input is a sequence $(d_1, g_1), (d_2, g_2), \cdots, (d_n, g_n)$ where g_i is a nonnegative real number representing the profit obtainable from job i, and $d_i \in \mathbb{N}$ is the deadline for job i; it doesn't hurt to assume that $1 \leq d_i \leq n$. (The reason why we can assume that every deadline is less than or equal to n is because even if some deadlines were bigger, every feasible schedule could be "contracted" so that no job was placed in a slot bigger than n.)

Definition 1 *A* schedule S *is an array:* $S(1), S(2), ..., S(n)$ *where* $S(t) \in \{0, 1, 2, \cdots n\}$ *for each* $t \in \{1, 2, \cdots, n\}$.

The intuition is that $S(t)$ is the job scheduled by S in slot t; if $S(t) = 0$, this means that no job is scheduled in slot t.

Definition 2 S *is* feasible *if*
(a) If $S(t) = i > 0$, *then* $t \leq d_i$. *(Every scheduled job meets its deadline)*
(b) If $t_1 \neq t_2$ *and* $S(t_1) \neq 0$, *then* $S(t_1) \neq S(t_2)$. *(Each job is scheduled at most once.)*

We define the *profit* of a feasible schedule S by
$P(S) = g_{S(1)} + g_{S(2)} + ... + g_{S(n)}$, where $g_0 = 0$ by definition.

Goal: Find a feasible schedule S whose profit $P(S)$ is as large as possible; we call such a schedule *optimal*.

We shall consider the following greedy algorithm. This algorithm begins by sorting the jobs in order of decreasing (actually nonincreasing) profits. Then, starting with the empty schedule, it considers the jobs one at a time; if a job can be (feasibly) added, then it is added to the schedule in the latest possible (feasible) slot.

Greedy:
Sort the jobs so that: $g_1 \geq g_2 \geq \ldots \geq g_n$
for $t : 1..n$
  $S(t) \leftarrow 0$ {Initialize array $S(1), S(2), ..., S(n)$}
end for
for $i : 1..n$
  Schedule job i in the latest possible free slot meeting its deadline;
  if there is no such slot, do not schedule i.
end for

Example. Input of **Greedy**:

Job i:	1	2	3	4	Comments
Deadline d_i:	3	2	3	1	(when job must finish by)
Profit g_i:	9	7	7	2	(already sorted in order of profits)

Initialize $S(t)$:

t	1	2	3	4
$S(t)$	0	0	0	0

Apply **Greedy**: Job 1 is the most profitable, and we consider it first. After 4 iterations:

t	1	2	3	4
$S(t)$	3	2	1	0

Job 3 is scheduled in slot 1 because its deadline $t = 3$, as well as slot $t = 2$, has already been filled.

$P(S) = g_3 + g_2 + g_1 = 7 + 7 + 9 = 23$.

Theorem 2 *The schedule output by the greedy algorithm is optimal, that is, it is feasible and the profit is as large as possible among all feasible solutions.*

We will prove this using our standard method for proving correctness of greedy algorithms.
We say feasible schedule S' *extends* feasible schedule S iff for all t $(1 \leq t \leq n)$,
if $S(t) \neq 0$ then $S'(t) = S(t)$.

Definition 3 *A feasible schedule is* promising after stage i *if it can be extended to an optimal feasible schedule by adding only jobs from* $\{i+1, \cdots, n\}$.

Lemma 2 *For* $0 \leq i \leq n$, *let* S_i *be the value of* S *after* i *stages of the greedy algorithm, that is, after examining jobs* $1, \cdots, i$. *Then the following predicate* $P(i)$ *holds for every* i, $0 \leq i \leq n$:

$P(i)$: S_i *is promising after stage* i.

This Lemma implies that the result of **Greedy** is optimal. This is because $P(n)$ tells us that the result of **Greedy** can be extended to an optimal schedule using only jobs from $\emptyset$. Therefore the result of **Greedy** must be an optimal schedule.

Proof of Lemma: To see that $P(0)$ holds, consider any optimal schedule S_{opt}. Clearly S_{opt} extends the empty schedule, using only jobs from $\{1, \cdots, n\}$.

So let $0 \leq i < n$ and assume $P(i)$. We want to show $P(i+1)$. By assumption, S_i can be extended to some optimal schedule S_{opt} using only jobs from $\{i+1, \cdots, n\}$.

Case 1: Job $i+1$ cannot be scheduled, so $S_{i+1} = S_i$.
Since S_{opt} extends S_i, we know that S_{opt} does not schedule job $i+1$. So S_{opt} extends S_{i+1} using only jobs from $\{i+2, \cdots, n\}$.

Case 2: Job $i+1$ is scheduled by the algorithm, say at time t_0 (so $S_{i+1}(t_0) = i+1$ and t_0 is the latest free slot in S_i that is $\leq d_{i+1}$).

Subcase 2A: Job $i+1$ occurs in S_{opt} at some time t_1 (where t_1 may or may not be equal to t_0).

Then $t_1 \leq t_0$ (because S_{opt} extends S_i and t_0 is as large as possible) and $S_{opt}(t_1) = i+1 = S_{i+1}(t_0)$.

If $t_0 = t_1$ we are finished with this case, since then S_{opt} extends S_{i+1} using only jobs from $\{i+2, \cdots, n\}$. Otherwise, we have $t_1 < t_0$. Say that $S_{opt}(t_0) = j \neq i+1$. Form S'_{opt} by interchanging the values in slots t_1 and t_0 in S_{opt}. Thus $S'_{opt}(t_1) = S_{opt}(t_0) = j$ and $S'_{opt}(t_0) = S_{opt}(t_1) = i+1$. The new schedule S'_{opt} is feasible (since if $j \neq 0$, we have moved job j to an earlier slot), and S'_{opt} extends S_{i+1} using only jobs from $\{i+2, \cdots, n\}$. We also have $P(S_{opt}) = P(S'_{opt})$, and therefore S'_{opt} is also optimal.

Subcase 2B: Job $i+1$ does not occur in S_{opt}.

Define a new schedule S'_{opt} to be the same as S_{opt} except for time t_0, where we define $S'_{opt}(t_0) = i+1$. Then S'_{opt} is feasible and extends S_{i+1} using only jobs from $\{i+2, \cdots, n\}$.

To finish the proof for this case, we must show that S'_{opt} is optimal. If $S_{opt}(t_0) = 0$, then we have $P(S'_{opt}) = P(S_{opt}) + g_{i+1} \geq P(S_{opt})$. Since S_{opt} is optimal, we must have $P(S'_{opt}) = P(S_{opt})$ and S'_{opt} is optimal. So say that $S_{opt}(t_0) = j$, $j > 0$, $j \neq i+1$. Recall that S_{opt} extends S_i using only jobs from $\{i+1, \cdots, n\}$. So

$j > i+1$, so $g_j \leq g_{i+1}$. We have $P(S'_{opt}) = P(S_{opt}) + g_{i+1} - g_j \geq P(S_{opt})$. As above, this implies that S'_{opt} is optimal. □

We still have to discuss the running time of the algorithm. The initial sorting can be done in time $O(n \log n)$, and the first loop takes time $O(n)$. It is not hard to implement each body of the second loop in time $O(n)$, so the total loop takes time $O(n^2)$. So the total algorithm runs in time $O(n^2)$. Using a more sophisticated data structure one can reduce this running time to $O(n \log n)$, but in any case it is a polynomial-time algorithm.

1.3 Approximating Simple Knapsack

In many situations, a greedy algorithm does not produce an optimal solution. But sometimes a greedy algorithm can give a reasonably good approximation for an otherwise hard (e.g., NP-hard) problem. Such is the case with the Simple Knapsack problem.

Informally, the problem is that we have a knapsack that can only hold weight C, and we have a bunch of items that we wish to put in the knapsack; each item has a specified weight, and the total weight of all the items exceeds C; we want to put items in the knapsack so as to come as close as possible to weight C, without going over. More formally, we can express the problem as follows. Let $w_1, \ldots, w_d \in \mathbb{N}$ be weights, and let $C \in \mathbb{N}$ be a weight. For each $S \subseteq \{1, \ldots, d\}$ let $K(S) = \sum_{i \in S} w_i$. (Note that $K(\emptyset) = 0$.)
Find:

$$M = \max_{S \subseteq \{1,\ldots,d\}} \{K(S) | K(S) \leq C\}$$

For large values of d, brute force search is not feasible because there are 2^d subsets of $\{1, \ldots, d\}$.

We can estimate M using the Greedy method:
We first sort the weights in decreasing (or rather nonincreasing order)

$$w_1 \geq w_2 \geq ... \geq w_d$$

We then try the weights one at a time, adding each if there is room. Call this resulting estimate $\overline{M}$.

It is easy to find examples for which this greedy algorithm does not give the optimal solution; for example weights $\{501, 500, 500\}$ with $C = 1000$. Then $\overline{M} = 501$ but $M = 1000$. However, this is just about the worst case:

Lemma 3 $\overline{M} \geq \frac{1}{2} M$

Proof: We first show that if $\overline{M} \neq M$, then $\overline{M} > \frac{1}{2} C$; this is left as an exercise. Since $C \geq M$, the Lemma follows. □

The notion of a **polynomial-time** algorithm is basic to complexity theory, and to this course (see definition below). Roughly speaking, we regard an algorithm as *feasible* (or *tractable*) if and only if it runs in polynomial time. The above greedy algorithm runs in polynomial time (see below) and is feasible to execute for values of d in the thousands or even millions. On the other hand, the blind search algorithm takes more than 2^d steps, is not polynomial-time, and will never run on any physical computer (now or in the future) for values of d as small as 200 – the universe will expire first.

Unfortunately the greedy algorithm does not necessarily yield an optimal solution. This brings up the question: is there any polynomial-time algorithm that is guaranteed to find an optimal solution to the simple knapsack problem? The answer is that this knapsack problem is "NP hard" (assuming that the weights are given using binary or decimal notation), and hence is very unlikely to be solvable by a polynomial-time algorithm.