CS 787: Advanced Algorithms 9/13/04

Lecture 2: Greedy Algorithms

Instructor: Dieter van Melkebeek Scribe: Marina Polishchuk, Ip Kei Sam

Today we will discuss the Greedy Method. An algorithm that uses the greedy method attempts to construct an optimal (compound) solution to a problem by optimally constructing components one by one. In general, building up locally optimal components may not yield a globally optimal overall solution, but a greedy algorithm will give an optimal result if these two coincide.

There are two main issues that arise when applying the greedy approach. First, in developing a greedy algorithm, the order in which we consider components of the problem is crucial. Second, we must also ensure the correctness of our greedy approach, i.e. we must prove that the greedy solution is an optimal one. We will discuss two different methods of proof that achieve these objectives.

1 Proof Method #1: Staying Ahead

1.1 Description:

The greedy method involves making a sequence of steps, where at each step a locally optimal choice with respect to the overall goal is made. Using a cost measure, we make sure that at any one of these steps, the greedy solution constructed up to that point is no worse than any other partial solution up to that point. In other words, at each step we've made so far, the greedy choice was one of possibly several optimal choices we could make. Formally, for any solution S and time step t, we show that the greedy partial solution at time t is at least as good as the corresponding restriction of S. Below are solutions to two problem instances using the greedy approach. Optimality is shown via the staying ahead proof technique.

1.2 Example: Interval Scheduling

Given: n jobs, specified by their time interval, i.e. $[start, end]$.

Goal: Schedule the maximum number of non-overlapping jobs on a single machine.

To apply the greedy approach, we must decide in which order to consider the jobs. One possibility would be to order these by doing the shortest job first. However, consider 3 jobs with times $1 : [0, 4.5], 2 : [5.5, 10]$, and $3 : [4, 6]$. The maximum number we can schedule is 2, jobs 1 and 3, but we will schedule only job 2 instead, ruling out both 1 and 3 as overlapping ones.

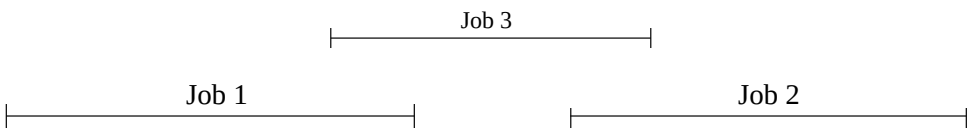

Figure 1: Interval Scheduling–Counterexample 1

Another idea would be to order the jobs by earliest start time. This fails for a similar reason, e.g. 3 jobs with times $1 : [0, 15], 2 : [1, 2]$, and $3 : [2, 3]$.

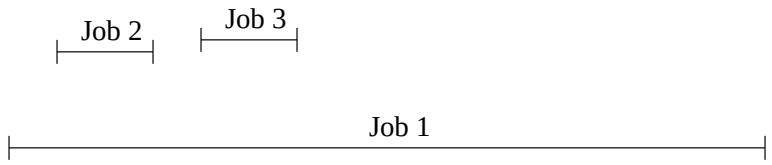

Figure 2: Interval Scheduling–Counterexample 2

We will show that ordering the jobs by earliest end time will produce an optimal solution.

Theorem 1. *The Earliest Finish Time First ordering generates an optimal schedule with respect to the goal. Formally, $\forall$ solution S, $\forall t \leq |S|$, the greedy algorithm schedules at least t jobs and finishes these t jobs at least as early as the first t jobs in S.*

Proof. We proceed by induction:

Base Case:

This is the case with $t = 0$: there is only one optimal solution, so the (vacuous) base case holds.

Inductive step:

Consider the step after we have scheduled the first t jobs. We'd like to show that, given the first t jobs produce an optimal schedule with respect to our criterion, our algorithm will produce an optimal schedule with $t + 1$ jobs.

Note that by assumption the end time of the first t jobs chosen by the greedy algorithm will be no worse (no later) than that of the first t jobs produced by some other approach. The next job chosen (job k below) is one with the next earliest end time, so the end time of our $t + 1$ jobs will be no later than the end time for any other solution's first $t + 1$ jobs.

□

The proof is illustrated below:

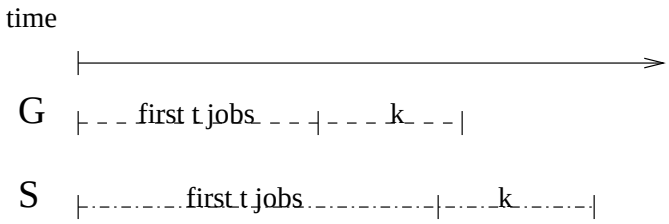

Figure 3: Interval Scheduling–maximum number of jobs scheduled

1.3 Example 2: Dijkstra's Shortest Path Algorithm

Given: A graph $G = (V, E)$ [either directed or undirected, with positive edge weights], and two vertices s and t.

Goal: Find the shortest path from s to t in G.

Description: Dijkstra's algorithm constructs a solution Z incrementally, starting with $Z = s$. At each step, we choose an additional vertex $u \in V - Z$ that is closest to some vertex $v \in Z$. In other words, the weight of edge (v, u) is less than the weight of any other edge (v', u') with $v' \in Z$ and $u \in V - Z$.

Notice the set Z contains vertices whose final shortest path weights from source s have already been computed. The algorithm repeatedly selects the vertex $u \in V - Z$ with the minimum shortest path estimate, inserts u into Z, and relaxes all edges leaving u. A priority queue Q is used to store all vertices in $V - Z$, keyed by their d values.

(Please refer to the ppt slides for this operation.)

Theorem 2. *The above approach produces the shortest path from s to t.*

Proof. Let P_u denote the path from s to u constructed by Dijkstra's algorithm. We will show that at any time, $\forall u \in Z, P_u$ is a shortest path from s to u. We argue by contradiction.

- Suppose there is an alternate path from s to u through v. Then, $pathlen(s, u) \geq label(v)$, where $label(v)$ denotes the cost of getting from s to v where all intermediate vertices are contained in Z.
- But $label(u) \leq label(v)$ (otherwise, we would not have chosen to add u to our solution Z). Since $pathlen(w, v) > 0$, we have a contradiction.

□

2 Proof Method #2: Exchange Arguments

2.1 Description:

Using a cost measure, we show that we can gradually transform any optimal solution O into our greedy solution K without making solution O any worse.

2.2 Example: Optimizing Procrastination

The problem of optimally scheduling tasks on a single processor, where each task has a duration and deadline, can be easily solved by a greedy algoritm.

Given: n jobs, specified by duration and deadline.

Goal: Schedule all jobs on a single machine in a non-overlapping way, so that maximum lateness is minimized. Here, we define lateness as follows: $lateness(i) = finishtime(i) - deadline(i)$.

This is illustrated Figure 4:

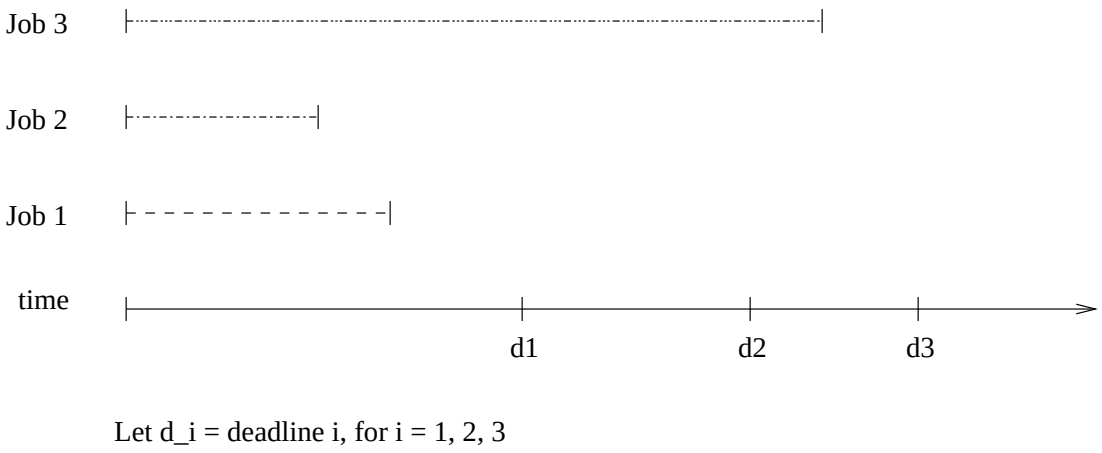

Figure 4: Interval Scheduling–minimizing maximum lateness

Theorem 3. *Scheduling the jobs using earliest deadline first (EDF) will yield an optimal solution, i.e. will minimize the maximum lateness.*

Motivation: Consider a given schedule S. We say that a task is late in this schedule if it finishes after its deadline. Otherwise, the task is early in the schedule. An arbitrary schedule can always be put into earliest deadline first, in which the tasks with early deadlines precede the tasks with late deadlines. That is, as long as there are two early tasks i and j finishing at respective times k and $k+1$, in the schedule, such that $d_j < d_i$, we swap the position of i and j.

Proof. First, we define the notion of an *inversion*. An inversion is a violation of the EDF rule in a solution S to the scheduling problem. For instance, if S schedules job k earlier than job m, but $deadline(k) > deadline(m)$. Notice that if we have a schedule that has an inversion, then swapping the two jobs involved eliminates the inversion at **no** cost.

Inversion Example: Consider a schedule that has an inversion: the first job scheduled has a later deadline than the second job (as illustrated in Figure 5).

Observation: The lateness of job 2 before swapping is larger than the lateness of job 1 after swapping. Since we moved job 2 to an earlier time, its lateness is not going to get worse. Also note that only the lateness of these two jobs changes when we swap. By swapping in this manner until there are no longer any inversions, we will indeed re-construct the solution produced by our greedy algorithm. □

Exercise: Does the greedy algorithm construct all optimal solutions? Try to prove that the optimal solutions produced are those that don't waste any time, use continuous scheduling, and have no inversions.

2.3 Example: Minimal Spanning Trees (Kruskal's algorithm)

Given: A weighted, undirected graph $G = (V, E)$. Let n be the number of vertices.

Goal: Find a spanning tree such that the sum of its weight is minimum.

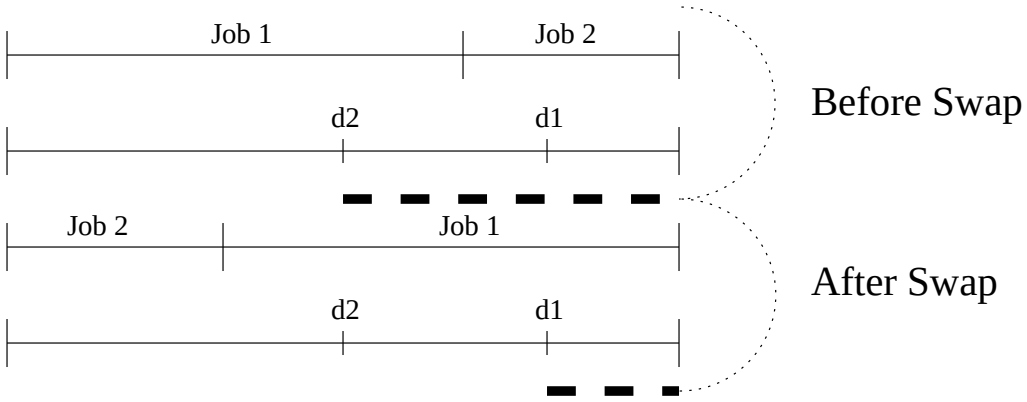

Figure 5: Interval Scheduling–swap procedure illustration

Theorem 4. *Starting with an empty tree, choosing the lowest weighted edge that does not add a cycle to the graph until we have chosen $n-1$ edges, will produce a minimal spanning tree (MST). This is Kruskal's algorithm.*

Proof. Consider any spanning tree T; let K = output of Kruskal, and assume that $T \neq K$. Then consider the first edge e in the order it got added to K where K decides differently than T. Note that $e \in K - T$; if it were in T, then T would contain a cycle. Adding e to T will induce a cycle C involving e. C contains an edge e' in $T - K$, because otherwise K would also have a cycle.

Now replace e' with e in T. Because $weight(e') \geq weight(e)$ (e' was not chosen by K prior to e), the overall weight will not become worse, and correctness is preserved. The new spanning tree T' has weight less than or equal to the weight of T and it is one step closer to K. Repeating this replacement process for the rest of the differing edges, we will arrive at solution K without increasing the weight of the spanning tree. Therefore, K is optimal. □

Analysis: The running time of Kruskal's algorithm for a graph $G = (V, E)$: initialization takes time $O(V)$, and the time to sort the edges is $O(E \cdot \log E)$. The algorithm involves $O(E)$ operations on the disjoint-set forest, which when implemented using Union-Find trees takes $O(E \cdot \alpha(E, V))$ time, where α is the functional inverse of the Ackermann function, with $alpha(E, V) = o(\log E)$. The total running time is dominated by the $O(E \cdot \log E)$ term. Please refer to Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms* for more details.