CS 787: Advanced Algorithms

Greed

Instructor: Dieter van Melkebeek

We start our review of undergraduate algorithm design with the greedy method.

1 Paradigm

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.

2 Proof Method #1: Staying Ahead

2.1 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.

2.2 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.

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 $3 : [2, 3]$.

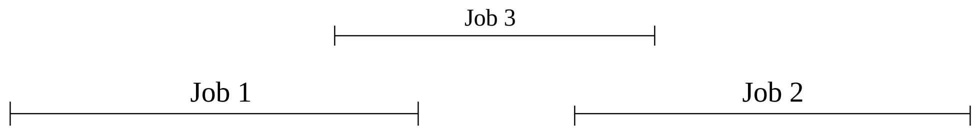

Figure 1: Interval Scheduling–Counterexample 1

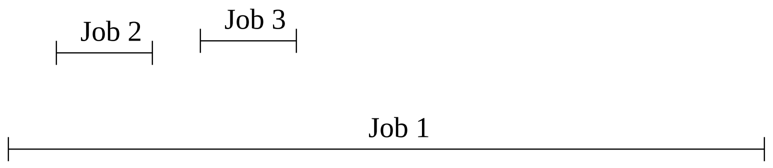

Figure 2: Interval Scheduling–Counterexample 2

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

In the following theorem we formalize a solution to the problem as a subset $S \subseteq [n] \doteq \{1, 2, \ldots, n\}$.

Theorem 1. *The Earliest Finish Time First ordering generates an optimal schedule with respect to the goal. Formally, for all solutions S and all non-negative integers $t \leq |S|$, 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 in the greedy schedule G is no worse (no later) than that of the first t jobs in S. This implies that the $(t + 1)$st job in S, say job i, is compatible with the first t jobs in G. Thus, the greedy schedule produces a $(t + 1)$st job, say job j, which may or may not equal i but definitely has a finish time no later than i by the greedy criterion. Thus, the induction step holds.

□

The proof is illustrated below:
The running time of the resulting greedy algorithm is $O(n \log n)$ because of the sorting.

2.3 Shortest Paths

Given: A graph $G = (V, E)$ [either directed or undirected] with non-negative edge lengths $\ell : E \to [0, \infty)$, and two vertices s and t.

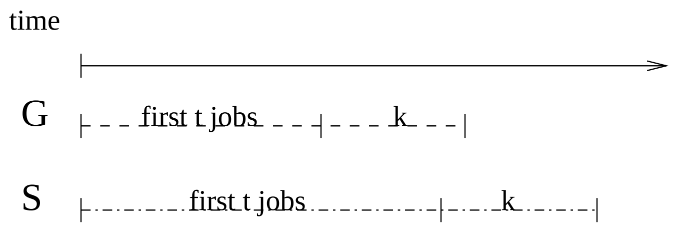

Figure 3: Interval Scheduling–maximum number of jobs scheduled

Goal: Find a shortest path from s to t in G, or report that there is no path from s to t.

Note the proviso that the edge lengths are non-negative. Later we will discuss a more general version of the problem where edge "lengths" can be negative. The special case where all edge lengths are the same can be solved using breadth-first search.

We define the distance $d(s,t)$ from s to t as the length of a shortest path from s to t if there is a path from s to t, and ∞ otherwise. Note that $\ell \geq 0$ guarantees that $d(s,t)$ is well-defined, but there may be multiple paths from s to t of length $d(s,t)$.

The greedy approach we consider is known as Dijkstra's algorithm. It gradually grows a set S of vertices $v \in V$ for which a path P_v of length $d(s,v)$ from s to v is constructed. We start with $S + \{s\}$ as the empty path realizes $d(s,s) = 0$. In each step we select an edge $(u,v) \in E$ with $u \in S$ and $v \in V \setminus S$ that minimizes the expression $d(s,u) + \ell(u,v)$. If no such edge exists, we report that there is no path from s to t. Otherwise we set P_v as the path P_u followed by the edge (u,v), and extend S with v. We keep growing S until it includes t (or we report that there is no path from s to t).

Theorem 2. *The above approach produces a shortest path from s to t if there exists a path from s to t.*

Proof. First note that if there is a path from s to t and t is not yet in S, then the edge (u,v) exists. It remains to establish the invariant that for every $v \in S$, P_v is a path from s to v of length $d(s,v)$.

Let $d^* \doteq d(s,u) + \ell(u,v)$, and consider any path P from s to v. Since P starts in S and ends outside of S, it contains at least one edge (u',v') such that $u' \in S$ and $v' \in V \setminus S$. By the choice of (u,v), the part of P from s to v' has length at least d^*. Since $\ell \geq 0$, the entire path P has length at least d^*. Since P_v goes from s to v and has length d^*, it is a shortest path from s to v. □

When implementating Dijkstra's algorithm, we keep track of a label $\lambda(v)$ for every $v \in V \setminus S$, which we set to the minimum value of $d(s,u) + \ell(u,v)$ over all $u \in S$ for which $(u,v) \in E$ (or ∞ if there is no such edge). We also keep track of which u realizes $\lambda(v)$. In each step we need to find the $v \in V \setminus S$ with the minimum value $\lambda(v)$, extend S by v, and update the labels of the vertices that can be reached in one step from v. The total number of label updates is bounded by the sum of the outdegrees, which equals $m \doteq |E|$. By keeping the vertices in $V \setminus S$ in a priority queue keyed on λ, the number of operations on the priority queue is at most $n + m$. In an efficient priority queue each operation only takes $O(\log n)$ operations. The resulting overall running time is $O(n+m)\log n)$. Compare this to the linear running time $O(n+m)$ for breadth-first search. A linear-time algorithm for shortest paths on graphs with non-negative edge lengths remains open.

Please refer to the presentation from class for a sample run of Dijkstra's algorithm.

3 Proof Method #2: Exchange Arguments

3.1 Description

We show that we can gradually transform any optimal solution O into our greedy solution G without every making the solution suboptimal. Each step involves a local modification (exchange) to the solution.

3.2 Minimizing Maximum Lateness

Given: n jobs, specified by duration and deadline d_i.

Goal: Schedule all jobs on a single machine starting at time $t = 0$ in a non-overlapping way so that $\max_{i \in [n]} \text{lateness}(i)$ is minimized, where $\text{lateness}(i) \doteq \max(0, \text{finishtime}(i) - d_i)$.

An example of an instance is given in Figure 4.

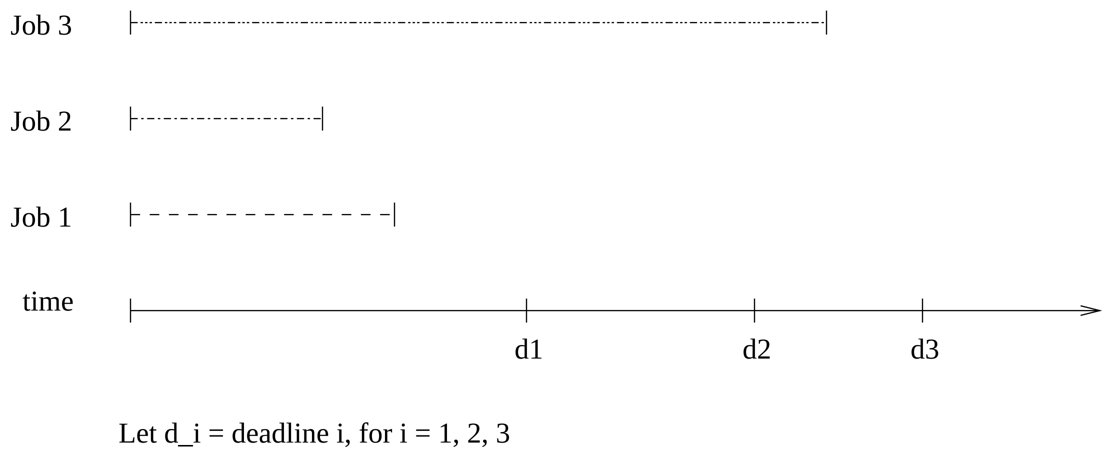

Figure 4: Interval Scheduling–minimizing maximum lateness

Note that the only real decision to make is the order in which to schedule the jobs. Earliest deadline first (EDF) turns out the be a good choice.

Theorem 3. *Scheduling the jobs back-to-back and earliest deadline first yields an optimal solution.*

Proof. We argue that any valid schedule can be transformed into the greedy EDF schedule G without increasing the maximum lateness.

First note that eliminating the gaps between jobs cannot increase the lateness of any job. This yields a solution in which jobs are scheduled back-to-back, but the ordering may deviate from EDF. If so, there have to be two jobs, say 1 and 2, such that 1 is scheduled right before 2 but $d_1 > d_2$. See the top of Figure 5) for an illustration.

Consider swapping jobs 1 and 2. This does not affect the lateness of the jobs other than 1 and 2. As job 2 is scheduled earlier, its lateness cannot increase. The lateness of job 1 can increase, namely when d_1 occurs before the finish time t^* of job 2 before the swap. This is the situation illustrated in Figure 5). However, note that the lateness of job 1 after the swap (the length of the line segment from d_1 to t^*) is no more than the lateness of job 2 before the swap (the length of the

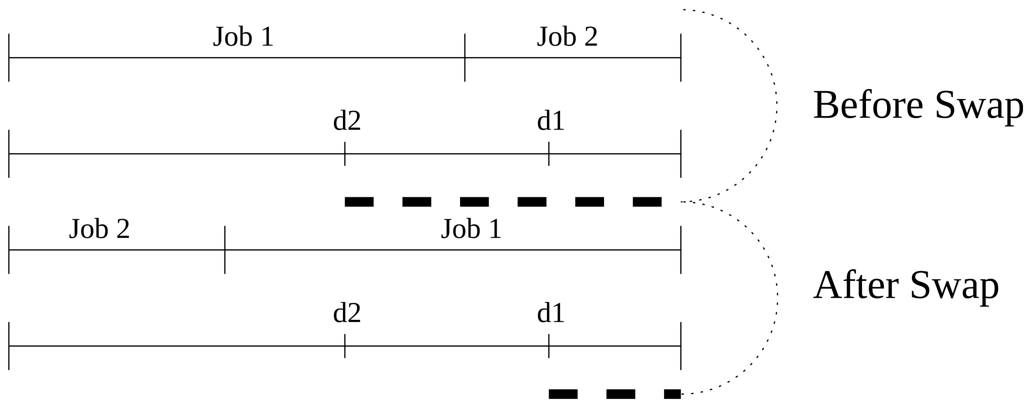
Job 1
Job 2
Before Swap
d2
d1
Job 2
Job 1
After Swap
d2
d1
Let ▬ ▬ denote lateness of job i
Let d_i denote the deadline of job i, for i = 1, 2

Figure 5: Interval Scheduling–swap procedure illustration

line segment from d_2 to t^*). This follows from the fact that $d_2 < d_1$. Thus, the maximum lateness over all jobs does not increase under the swap.

The swap operation brings us closer to the greedy solution in the following precise sense. Define an *inversion* as a pair of jobs that are scheduled in violation with the EDF rule, i.e., the job that is scheduled earlier has a later deadline. Note that the above swap operation reduces the number of inversions by one. Since the number of inversion is at most $\binom{n}{2}$, after a finite number of swaps we reach a schedule with no inversion, i.e., the greedy schedule.

Thus, the maximum lateness of the greedy schedule is no more than of any valid schedule, i.e., the greedy schedule is optimal. □

The running time of the resulting greedy algorithm is $O(n \log n)$ because of the sorting.

3.3 Minimum Spanning Trees

Given: An undirected connected graph $G = (V, E)$ with edge weights $w : E \to (-\infty, \infty)$. vertices.

Goal: Find a spanning tree such that the sum of weight of the edges is minimized.

There are several greedy algorihtms for constructing minimum spanning trees. One of them goes over the edges in order of non-increasing weight and includes an edge in the tree unless it induces a cycle in the subgraph constructed thus far. The algorithm is known as Kruskal's algorithm. A more descriptive name would be "tree-unioning algorithm", as it can be viewed as starting from n trees of size 1 and in each step unions two of them via an edge.

Theorem 4. *Kruskal's algorithm produces a minimum spanning tree.*

Proof. Let K denote the subgraph returned by Kruskal's algorithm. By construction K contains no cycles. The connectivity of G implies that K is also connected. As K contains all vertices of G, we have that K is a spanning subtree of G. All that remains is to argue the minimality of K.

We use an exchange argument to do so. Start from any spanning tree T of G, and assume $T \neq K$. Then consider the first edge e in the greedy order that is in the symmetric difference of K and T. Note that e cannot be in T; if it were then T would contain a cycle, namely the cycle that caused Kruskal's algorithm not to include e in K. Thus, e is in K but not in T. Adding e to T would induce a cycle C involving e. C contains an edge e' that is in T but not in K; otherwise K would also contain the cycle C. Let T' be T with e' replaced by e.

- T' remains connected because it contains a path between the two endpoints of the removed edge e', namely C with e' removed. As the number of edges in T and T' is the same, it follows that T' is a spanning tree of G.

- As e was the first edge in the symmetric difference of K and T, e' appears further down the greedy order than e so $w(e') \geq w(e)$. Since $w(T') - w(T) = w(e) - w(e')$, it follows that $w(T') \leq w(T)$.

Note that the symmetric difference between K and T' is smaller than between K and T. Thus, after a finite number of such exchange operations the resulting spanning subtree has to coincide with K. As the weight has never increased, the weight of K is no more than of any spanning subtree of G. Hence, K is optimal. $\square$

Kruskal's algorithm needs to check whether an edge $e = (u, v)$ induces a cycle in the subgraph constructed thus far. We do so using a union-find data structure, which maintains the list of connected components of the subgraph and allows two type of operations: merging two of the connected components (union) and asking the label of the component of a given vertex (find). Kruskal's algorithm issues $n-1$ unions and at most m finds. There exists a union-find data structure for which the total amount of work for these operation is $O(m \cdot \alpha(m))$ time. Here α denotes the inverse Ackerman function, which is extremely slowly growing and can be considered constant for all practical purposes. A simpler implementation executes a union operation by relabeling the vertices in the smaller connected component by the label of the larger one; for that data structure the total amount of work for the union-find operations is $O(m \log m)$. As the sorting of the edges by their weight takes time $O(m \log m)$, in either case the resulting running time is $O(m \log m)$.

Another greedy algorithm for constructing minimum spanning trees is the tree-growing algorithm known as Prim's algorithm. The algorithm has some similarity with Dijkstra's algorithm in that it grows a set S of vertices on which a minimum spanning tree is known. In each step, it adds an edge connecting S to $V \setminus S$ of minimum weight to the tree. A similar exchange argument argues optimality, and an efficient implementation using priority queues also runs in time $O(m \log m)$. Please refer to the presentation from class for a sample runs of Kruskal's and Prim's algorithm.