Adapted From Virginia Williams' lecture notes.

COMP 285 (NC A&T, Spr '22) Lecture 28

Getting Greedy with Algorithms

1 Greedy Algorithms

Suppose we want to solve a problem, and we're able to come up with some recursive formulation of the problem that would give us a nice dynamic programming algorithm. But then, upon further inspection, we notice that any optimal solution only depends on looking up the optimal solution to one other subproblem. A greedy algorithm is an algorithm which exploits such a structure, ignoring other possible choices. Greedy algorithms can be seen as a refinement of dynamic programming; in order to prove that a greedy algorithm is correct, we must prove that to compute an entry in our table, it is sufficient to consider at most one other table entry; that is, at each point in the algorithm, we can make a "greedy", locally-optimal choice, and guarantee that a globally-optimal solution still exists. Instead of considering multiple choices to solve a subproblem, greedy algorithms only consider a single subproblem, so they run extremely quickly – generally, linear or close-to-linear in the problem size.

Unfortunately, greedy algorithms do not always give the optimal solution, but they frequently give good (approximate) solutions. To give a correct greedy algorithm one must first identify optimal substructure (as in dynamic programming), and then argue that at each step, you only need to consider one subproblem. That is, even though there may be many possible subproblems to recurse on, given our selection of subproblem, there is always an optimal solution that contains the optimal solution to the selected subproblem.

1.1 Activity Selection

One problem, which has a very nice (correct) greedy algorithm, is the Activity Selection Problem.

In this problem, we have a number of activities. Your goal is to choose a subset of the activities to participate in. Each activity has a start time and end time, and you can't participate in multiple activities at once. Thus, given n activities $a_1, a_2, \cdots, a_n$ where a_i has start time s_i and finish time f_i , we want to find a maximum set of non-conflicting activities.

The activity selection problem has many applications, most notably in scheduling jobs to run on a single machine.

1.1.1 Optimal Substructure

Let's start by considering a subset of the activities. In particular, we'll be interested in considering the set of activities $S_{i,j}$ that start after activity a_i finishes and end before activity

a_j starts. That is, $S_{i,j} = \{a_k \mid f_i \leq s_k, f_k \leq s_j\}$. We can participate in these activities between a_i and a_j. Let $A_{i,j}$ be a maximum subset of non-conflicting activities from the subset $S_{i,j}$. Our first intuition would be to approach this by using dynamic programming. Suppose some $a_k \in A_{i,j}$, then we can break down the optimal subsolution $A_{i,j}$ as follows

$$|A_{i,j}| = 1 + |A_{i,k}| + |A_{k,j}|$$

where $A_{i,k}$ is the best set for $S_{i,k}$(before a_k), and $A_{k,j}$ is the best set for after a_k . Another way of interpreting this expression is to say "once we place a_k in our optimal set, we can only consider optimal solutions to subproblems that do not conflict with a_k ."

Thus, we can immediately come up with a recurrence that allows us to come up with a dynamic programming algorithm to solve the problem.

$$|A_{i,j}| = \max_{a_k \in S_{i,j}} 1 + |A_{i,k}| + |A_{k,j}|$$

.

This problem requires us to fill in a table of size n^2, so the dynamic programming algorithm will run in $\Omega(n^2)$ time. The actual runtime is $O(n^3)$ since filling in a single entry might take $O(n)$ time.

But we can do better! We will show that we only need to consider the a_k with the smallest finishing time, which immediately allows us to search for the optimal activity selection in linear time.

Proposition 1. For each $S_{i,j}$, there is an optimal solution $A_{i,j}$ containing $a_k \in S_{i,j}$ of minimum finishing time f_k.

Note that if the proposition is true, when f_k is minimum, then $A_{i,k}$ is empty, as no activities can finish before a_k ; thus, our optimal solution only depends on one other subproblem $A_{k,j}$ (giving us a linear time algorithm).

Here, we prove the proposition.

Proof. Let a_k be the activity of minimum finishing time in $S_{i,j}$. Let $A_{i,j}$ be some maximum set of non-conflicting activities. Consider $A'_{i,j} = A_{i,j} \setminus a_l \cup a_k$ where a_l is the activity of minimum finishing time in $A_{i,j}$. It's clear that $|A'_{i,j}| = |A_{i,j}|$. We need to show that $A'_{i,j}$ does not have conflicting activities. We know $a_l \in A_{i,j} \subset S_{i,j}$. This implies $f_l \geq f_k$, since a_k has the minimum finishing time in $S_{i,j}$.

All $a_t \in A_{i,j} \setminus a_l$ don't conflict with a_l, which means that $s_t \geq f_l$, which means that $s_t \geq f_k$, so this means that no activity in $A_{i,j} \setminus a_l$ can conflict with a_k . Thus, $A'_{i,j}$ is an optimal solution. □

Due to the above proposition, the expression for $A_{i,j}$ from before simplifies to the following expression in terms of $a_k \subseteq S_{i,j}$, the activity with minimum finishing time f_k .

$$|A_{i,j}| = 1 + |A_{k,j}|$$
$$A_{i,j} = A_{k,j} \cup \{a_k\}$$

Algorithm Greedy-AS assumes that the activities are presorted in nondecreasing order of their finishing time, so that if $i < j$, $f_i \le f_j$.

Algorithm 1: Greedy-AS(a)

```
$A \leftarrow \{a_1\}$ /* activity of min $f_i$
$k \leftarrow 1$
for $m = 2 \rightarrow n$ do
    if $s_m \ge f_k$ then
        // $a_m$ starts after last activity in A
        $A \leftarrow A \cup \{a_m\}$
        $k \leftarrow m$
return $A$
```

By the above claim, this algorithm will produce a legal, optimal solution via a greedy selection of activities. There may be multiple optimal solutions, but there always exists a solution that includes a_k with the minimum finishing time. The algorithm does a single pass over the activities, and thus only requires $O(n)$ time – a dramatic improvement from the trivial dynamic programming solution. If the algorithm also needed to sort the activities by f_i , then its runtime would be $O(n \log n)$ which is still better than the original dynamic programming solution.