CSE347 Class 3

Jeremy Buhler

September 6, 2017

1 The Class of Optimization: Problems

Today, we will focus on *combinatorial optimization* problems.

- A problem instance includes a set of *constraints* (properties that a valid solution must satisfy) and an *objective function*, or measure of goodness/badness, for solutions.
- Constraints are typically discrete rather than continuous (no such thing as an "almost correct" answer)
- Goal is to find a solution that is
  - *feasible* – satisfies all constraints
  - *optimal* – maximizes (or minimizes) objective
- We want an efficient algorithm that finds an optimal feasible solution for any problem instance.
- Note that this is a bit different paradigm than the algorithms we've seen so far – instances of GCD or matrix multiplication have just one correct answer, so we had to prove that the answer returned by Euclid or Strassen is the right one.
- In contrast, optimization demands that we consider a large space of feasible solutions and return one that is best according to the objective.
- To prove an optimization algorithm correct, we need to show that the solution it returns is not only feasible but *optimal* among all feasible solutions.

2 A Scheduling Problem

To begin our study of combinatorial optimization, let's study a concrete example.

- You manage a ginormous space telescope.
- Lots of astronomers want to use it to make observations.
- Each astronomer's project p_i requires use of the telescope starting at a fixed time s_i (when their grant starts) and running for ℓ_i days.
- Only one project can use the telescope at a time.

- Your goal: justify your budget to NASA by scheduling as many projects as possible!
- More formally: given a set P of projects p_i, each occupying half-open time interval $[s_i, s_i + \ell_i)$,
- Choose a subset $\Pi \subseteq P$ of projects for which
  - No two projects' intervals overlap ("conflict");
  - The number of projects in Π is maximized.
- This is one of many variants of the *scheduling* or *activity selection* problem.

Example

How should we solve this problem?

- We can't afford to enumerate all possible subsets of projects, because a set of n projects could have $\Theta(2^n)$ feasible subsets.
- So maybe we can find some rule for building up a solution that is guaranteed to be optimal.
- **Suggestion 1**: repeatedly pick shortest non-conflicting, unscheduled project (i.e. that does not conflict with any scheduled project).
- Does this strategy always yield an optimal solution? Prove or disprove.
- **Counterexample**:

- **Suggestion 2**: repeatedly pick non-conflicting project with earliest starting time.
- Does this always yield an optimal solution? Prove or disprove.
- **Counterexample**:

- **Suggestion 3**: first, label each project with number of *other* projects with which it conflicts. Then, repeatedly pick non-conflicting project with fewest total conflicts.
- Does this always yield an optimal solution? Prove or disprove.

- **Counterexample**:

Aaaaargh! As before, we need a *principle* to guide development of an algorithm that is provably optimal.

3 An Approach That Works

What structure do all above solutions have in common?

- Repeatedly pick an element until no more feasible choices remain.
- Among all feasible choices, we always pick the one that minimizes or maximizes some property (project length, start time, # conflicts)
- Such algorithms are called *greedy*.
- As we've seen, greedy algorithms are frequently *not* optimal.
- Ah, but maybe we have been using the wrong property!

4 A Greedy Choice that Leads to Optimality

Let's take another wild guess...

- For each project p_i, define its *finishing time* f_i to be $s_i + \ell_i$.
- *Repeatedly pick non-conflicting, unscheduled project with earliest finishing time.*
- Does this algorithm always produce a feasible solution?
- Yup – we always pick the *non-conflicting* project with earliest scheduled end time, so we never pick two projects that conflict.
- Does it yield optimal solutions on our "bad cases" from above? Yes!
- So, can we argue that this approach always produces an optimal solution?
- There is no guarantee that there is a *unique* optimal solution to any given problem instance. So we can't show that our algorithm finds "the answer."
- But perhaps we can show that its answer is *at least as good as any other* – hence optimal.

- Let Π be the solution obtained by our greedy algorithm, and let Π_0 be some other feasible solution.
- Let's say that Π contains k projects with intervals $[s_{i_1}, f_{i_1}) \ldots [s_{i_k}, f_{i_k})$, sorted in increasing order of finishing time (eqv. of starting time).
- Similarly, say that Π_0 contains m projects with intervals $[s_{j_1}, f_{j_1}) \ldots [s_{i_m}, f_{j_m})$, again sorted in increasing order of finishing time.

We're going to prove that $k \geq m$ (i.e., $|\Pi| \geq |\Pi_0|$, and hence Π is always at least as good as Π_0 relative to our objective).

- Suppose we could prove the following key lemma:

  > For all $r \leq \min(k, m)$, the first r projects in Π (in order by finishing times) always finish in the same or less time than the first r projects in Π_0; that is, $f_{i_r} \leq f_{j_r}$.

- Informally, the greedy solution "stays ahead" of the other solution, in that the first r projects of Π fit in a shorter interval than the first r projects of Π_0.
- I claim that this lemma implies $|\Pi| \geq |\Pi_0|$.
- Suppose not, i.e., suppose that $|\Pi| < |\Pi_0|$ (in other words, $k < m$).
- Consider the $k+1$st project in Π_0. By our lemma, the first k projects of Π (i.e. all of Π) finish no later than the first k projects of Π_0.
- Hence, $s_{j_{k+1}} \geq f_{j_k} \geq f_{i_k}$.
- Conclude that we could add the $k+1$st project from Π_0 to Π and obtain a bigger, hence *better*, feasible solution.
- But our greedy algorithm will always add a non-conflicting project to the solution if it is possible to do so!
- Hence, Π would already contain this $k+1$st project, so cannot have size k as claimed. $\rightarrow\leftarrow$.
- Therefore, it must be that $k \geq m$. QED

OK, now how do we prove the lemma?

- The lemma most hold for every $r > 0$, so we'll proceed by induction on r.
- **Bas**: when $r = 1$, we have by construction that the greedy algorithm picks a project with $f_{i_1} \leq f_{j_1}$.
- **Ind**: suppose that the lemma holds for the first r projects in each of Π and Π_0.
- Then by the IH, $f_{i_r} \leq f_{j_r}$.
- Hence, every remaining project that can be feasibly added to Π_0 can also be added to Π.

- Π_0 contains one of these projects,namely $p_{j_{r+1}}$.
- But Π contains the particular project $p_{i_{r+1}}$ with earliest finishing time.
- Conclude that $f_{i_{r+1}} \leq f_{j_{r+1}}$, as desired. QED

5 A General Idea

- The above is an example of a design/proof strategy called "greedy stays ahead."
- We want to prove that the greedy algorithm's solution Π has at least as good an objective value as any other feasible solution Π_0.
- To do so, we find a way in which the partial solutions built up by the greedy algorithm "stay ahead" of the parts of any arbitrary solution.
- In our example, the "stay ahead" property was the time at which the last job in the solution finished.
- As we saw, proving the "stay ahead" property allowed us to argue that the full greedy solution Π is optimal – otherwise, it could "borrow" an element from any better (bigger) solution to improve itself.
- The trick is finding a "stay ahead" property which is both true for every partial solution (prove by induction) *and* helpful in proving optimality.
- Design your algorithm's greedy choice to make the proof possible!
- *Hint*: *first* find a "stay ahead" property that implies optimality (in our case, that each prefix of the greedy solution finishes in the least possible time), *then* pick a greedy choice to ensure this property.

6 Efficient Implementation

- To implement our algorithm efficiently, we need to be able to repeatedly make the greedy choice quickly.
- In this case, the choice is the *non-conflicting* project with the *earliest finishing time.*
- *Idea*: first, sort all projects in non-decreasing order by finishing time.
- Repeatedly pick the first project that does not conflict with those already chosen.
- A project conflicts if it overlaps the last project chosen.
- To be a bit more precise, here is some pseudocode:

```
SCHEDULE(P)
  sort P in increasing order {p_1 ... p_n} of finishing time f_i
  Π ← {p_1}
  j ← 1
  for i in 2..n do
      if s_i ≥ f_j
          Π ← Π ∪ {p_i}
          j ← i
  return Π
```

- Sorting the projects requires $O(n \log n)$ time if we use some efficient sort, such as a mergesort.
- The selection loop takes $O(1)$ time for each project i, in the list, so $O(n)$ time overall.
- Hence, total complexity of SCHEDULE is $O(n \log n)$.

7 A Challenge for Next Time

- Suppose that not all projects are equally important.
- In particular, each project p_i has a non-negative *importance* w_i.
- NASA would now like you to maximize the total *importance* of all projects scheduled.
- Clearly, maximizing the number of projects need not maximize their total importance.
- Can you come up with an efficient algorithm to solve this problem optimally?