Greedy Algorithms

CS 5633 Analysis of Algorithms

Computer Science
University of Texas at San Antonio

November 4, 2024

Greedy Algorithm Basics

Optimization Problems

- An *optimization problem* is a problem in which we want to find a “best” solution out of a set of *feasible* solutions.
  - A solution is feasible if it satisfies a given set of constraints.
- Generally an optimization problem will want to maximize some value or minimize some cost.
  - Maximization example: Toll Booth problem.
  - Minimization example: Matrix Chain.
- We have considered some dynamic programming approaches to some optimzation problems. For some problems, dynamic programming is overkill, and there may be a more efficient algorithm.

Greedy Algorithms

- A **greedy algorithm** is an algorithm which solves a subproblem by making a choice which appears to be the best choice at the moment. After making this choice, we obtain a new subproblem which we again solve by making a choice which appears to be the best choice at that moment.
  - Matrix Chain example: Find the largest p_i and multiply A_i with A_{i+1}. Repeat this procedure until there is only one matrix.
  - Toll Booth example: Choose the toll booth with the largest value, and remove any toll boths which would be too close to this booth. Repeat this procedure until we cannot add any more toll booths.
- Neither one of these examples guarantees an optimal solution. That is, there are *counterexamples* for which these algorithms will compute a solution which is not optimal.

Greedy Algorithms cont.

- That being said, there are many problems for which there exist greedy algorithms which guarantee to return an optimal solution, and generally a greedy algorithm will be more efficient (in time and space) than a dynamic programming algorithm.
- For example, the gradient descent algorithm in Machine Learning is a typical greedy algorithm. It will always find the optimal solution if the targeted function is a convex function.

The Activity-Selection Problem

The Activity-Selection Problem

- Suppose we have a set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed *activities* that wish to use a resource (e.g. a lecture hall) which can serve only one activity at a time.
- Each activity a_i has a *start time* s_i and a *finish time* f_i where
$$0 \leqslant s_i < f_i < \infty$$
- If selected, activity a_i takes place during the half-open time interval $[s_i, f_i)$. Activities a_i and a_j are *compatible* if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.
- The **activity-selection problem** wants to select a maximum-size subset of mutually compatible activities.

An Example of Activity-Selection Problem

- We assume that the activities are listed in non-decreasing order according to finishing time:

$$f_1 \leqslant f_2 \leqslant \ldots \leqslant f_n$$

- Activities a_1, a_4, a_8 and a_{11} constitute the largest subset of mutually compatible activities.

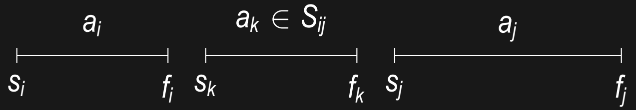
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

The Dynamic Programming Solution to Activity-Selection Problem

Activities to Intervals

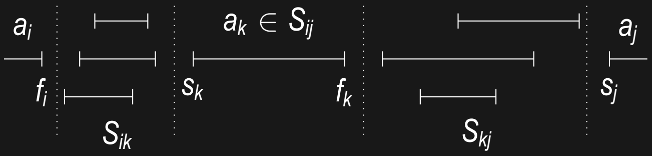
- Note we can view the activities as intervals where the left end is s_i and the right end is f_i.

- Let S_{ij} denote the set of activities that start after a_i finishes and finish before a_j starts. Suppose we wish to find a maximum set of mutually compatible activities in S_{ij}.



Constructing Recursive Algorithm

- Let A_{ij} denote the optimal solution, and suppose it contains some activity a_k. Note that a_k divides S_{ij} into two subproblems S_{ik} and S_{kj}.
- The optimal solution for S_{ij} is therefore $A_{ik} \cup A_{kj} \cup \{a_k\}$. The size of the solution is $|A_{ik}| + |A_{kj}| + 1$.



Constructing Recursive Algorithm

- Let $c[i,j]$ denote the size of an optimal solution for S_{ij}. We have the following recurrence relation (note this is similar to the matrix chain problem):

$$c(i,j) = \begin{cases} 0, & \text{if } S_{ij} = \phi \\ \max\limits_{a_k \in S_{ij}} (c[i,k] + c[k,j] + 1), & \text{if } S_{ij} \neq \phi \end{cases}$$

- We could solve this problem via dynamic programming, and the running time would be $O(n^3)$ (we have $O(n^2)$ subproblems, and each one takes $O(n)$ time to compute). Can we do better?

Intuition of the Greedy Solution

- What might be a good greedy strategy when determining an activity to include in our optimal solution?
- Intuition tells us that it may be a good idea to include an activity which ends as early as possible, as we increase the number of activities in our solution and we leave as much time left as possible for the remaining activities.
- Could it be that repeating this procedure repeatedly until we cannot add any more activities results in an optimal solution? Can we either prove that the algorithm is correct or can we construct a counterexample which shows that it fails?

Formal Proof of the Greedy Solution

- Let $S_k = \{a_i \in S : s_i \geq f_k\}$ be the set of activities that start after a_k finishes. We will prove the following theorem.
- **Theorem**: Consider any nonempty sub-problem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.
- **Proof**:
  - Let A_k be a maximum-size subset of mutually compatible activities in S_k.
  - Let a_j be the activity in A_k with the earliest finish time.
  - If $a_j \neq a_m$, we can always construct a new set A'_k by removing a_j from A_k and adding a_m to it. That is $A'_k = (A_k - \{a_j\}) \cup \{a_m\}$.
  - Clearly A'_k is also an optimal solution as: 1) $|A'_k| = |A_k|$; 2) none of the activities in A'_k starts before a_m finishes.

The Greedy Algorithm

Algorithm 1: Greedy algorithm for the activity-selection problem.

Function Greedy_Activity_Selector(start_times s, finish_times f)

  // Assuming activities are ordered by monotonically increasing finish time;

  $A = \{a_1\}$; // the activity that finishes first is always in the optimal solution;

  $k = 1$;

  // go over the activities one by one with increasing finish time;

  for m=2 **to** $s.length$ **do**

    if $s[m] \leqslant f[k]$ **then**

      // if a_m starts after all activities in A, add a_m to A;

      $A = A \cup \{a_m\}$;

      // let a_m be a_k, the activity in A that finishes last;

      k = m;

  return A;

Run Time of the Greedy Algorithm

- The Greedy_Activity_Selector has a run time of $\Theta(n)$.
- Sorting the activities has a cost of $\Theta(n \lg n)$.
- Even with sorting, the greedy algorithm has a much better run time than the dynamic programming algorithm.

Considerations for Greedy Algorithms

- Greedy algorithms depend on a strategy that always chooses the best choice at current moment.
- The currently-best choice may not always be obvious. A good understanding of the problem is usually required.
- The difficult part is to prove the algorithm's correctness. That is, by always choosing the currently-best choice, it is possible to get an (global) optimal solution.
- In some cases, a greedy algorithm may stuck on a local optimal solution than the global optimal solution.
- For many problems, finding the global optimal may not be feasible (e.g., NP-complete problems). Therefore, greedy algorithms are usually employed as heuristics to find local optimal solutions, which may be only slightly worse than the global optimal.