Greedy Algorithms

Chapter 16

Greedy Algorithms

- A greedy algorithm always makes the choice that looks best at the moment
- Greedy algorithms do not always lead to optimal solutions, but for many problems they do

Activity-Selection Problem

- You are given a list of programs to run on a single processor
- Each program has a start time and a finish time
- The processor can only run one program at any given time, and there is no preemption
- **Goal:** Select the largest possible set of nonoverlapping (*mutually compatible) activities.*

Other examples: scheduling a lecture hall, and deciding which movies to star in to make as much money as possible ☺

Example

- S is a set of activities sorted by finishing time $S = \{a_1, a_2, a_3, \ldots, a_n\}$:

i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16

- Maximum-size mutually compatible set:
{ }

Dynamic Programming

- How would we solve the activity-selection problem using dynamic programming?

Optimal Substructure

$$S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$$

- Activities in S_{ij} are compatible with
  - All activities that finish by f_i, and
  - All activities that start no earlier than s_j.

Optimal Substructure (cont.)

- Let A_{ij} be a maximum-size set of mutually compatible activities in S_{ij}.
- Let $a_k \in A_{ij}$ be some activity in A_{ij}. Then we have two suproblems:
  - Find mutually compatible activities in S_{ik}
  - Find mutually compatible activities in S_{kj}

Optimal Substructure (cont.)

- Let: $A_{ik} = A_{ij} \cap S_{ik}$= activities in A_{ij} that finish before a_k starts,
- $A_{kj} = A_{ij} \cap S_{kj}$ = activities in A_{ij} that start after a_k finishes.
- Then $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$
- $\Rightarrow |A_{ij}| = |A_{ik}| + |A_{kj}| + 1$
- **Claim:** Optimal solution A_{ij} must include optimal solutions for the two subproblems for S_{ik} and S_{kj}

Recursive Solution

- Let c[i,j] = size of optimal bolution for S_{ij}. Then:
  - c[i,j] = c[i,k] + c[k,j] + 1
- But, we don't know which activity a_k to choose, so we have to try them all:

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

Alternative Approach (Greedy)

- Choose an activity to add to optimal solution before solving subproblems. For activity-selection problem, we can get away with considering only the greedy choice: the activity that leaves the resource available for as many other activities as possible.
- Question: Which activity leaves the resource available for the most other activities?

Optimal Substructure

- Since we only have one subproblem to solve, we simplify notation:

$$S_k = \{a_i \in S : s_i \geq f_k\}$$

- By optimal substructure, if a_1 is in an optimal solution, then an optimal solution to the original problem consists of a_1 plus all activities in an optimal solution to S_1. But need to prove that a_1 is always part of some optimal solution.

Greedy Solution

- So, don't need full power of dynamic programming. Don't need to work bottom-up.
- Instead, can just repeatedly choose the activity that finishes first, keep only the activities that are compatible with that one, and repeat until no activities remain.
- Can work top-down: make a choice, then solve a subproblem. Don't have to solve subproblems before making a choice.

Recursive Greedy Algorithm

- Start and finish times are represented by arrays s and f , where f is assumed to be already sorted in monotonically increasing order.
- To start, add fictitious activity a_0 with $f_0 = 0$, so that $S_0 = S$, the entire set of activities.

Recursive Algorithm

REC-ACTIVITY-SELECTOR(s, f, k, n)
$m = k + 1$
while $m \le n$ and $s[m] < f[k]$ // find the first a
    $m = m + 1$
if $m \le n$
    return $\{a_m\} \cup$ REC-ACTIVITY-SELECTOR(s, f, m, n)
else return $\emptyset$

Initial call

REC-ACTIVITY-SELECTOR$(s, f, 0, n)$.

Iterative Algorithm

GREEDY-ACTIVITY-SELECTOR(s, f)
$n = s.length$
$A = \{a_1\}$
$k = 1$
for $m = 2$ **to** n
    if $s[m] \ge f[k]$
        $A = A \cup \{a_m\}$
        $k = m$
return A

How Did We Solve?

- Determine the optimal substructure.
- Develop a recursive solution.
- Show that if we make the greedy choice, only one subproblem remains.
- Prove that it's always safe to make the greedy choice.
- Develop a recursive greedy algorithm.
- Convert it to an iterative algorithm.

Typically

- Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
- Prove that there's always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.
- Demonstrate optimal substructure by showing that, having made the greedy choice, combining an optimal solution to the remaining subproblem with the greedy choice gives an optimal solution to the original problem.

Greedy-Choice Property

- Can assemble a globally optimal solution by making locally optimal (greedy) choices.

Dynamic Programming	**Greedy**
•Make a choice at each step. •Choice depends on knowing optimal solutions to subproblems. Solve subproblems first. •Solve bottom-up.	•Make a choice at each step. •Make the choice before solving the subproblems. •Solve top-down.

Greedy vs. Dynamic Programming

- 0-1 Knapsack problem
  - n items.
  - Item i is worth $i , weighs w_i pounds.
  - Find a most valuable subset of items with total weight W .
  - Have to either take an item or not take it—can't take part of it.
- Fractional Knapsack problem
  - Like the 0-1 knapsack problem, but can take a fraction of an item.

Greedy Solution

FRACTIONAL-KNAPSACK(v, w, W)

```
load = 0
i = 1
while load < W and i ≤ n
    if w_i ≤ W − load
        take all of item i
    else take (W − load)/w_i of item i
    add what was taken to load
    i = i + 1
```


0-1 Knapsack Problem

- Is there a greedy solution?
- Example:

i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i/w_i	6	5	4

$W = 50.$