CMPT307: Greedy Algorithms

Week 8-1

Xian Qiu

Simon Fraser University

xianq@sfu.ca

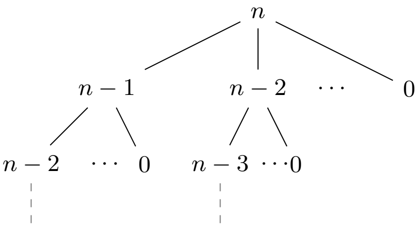
Outline

- remarks on dynamic programming
- greedy algorithms

Overlapping Subproblems

recall the rod cutting problem and its recursive formula

$$r_n = \max_{1 \le i \le n} \{p_i + r_{n-i}\}$$



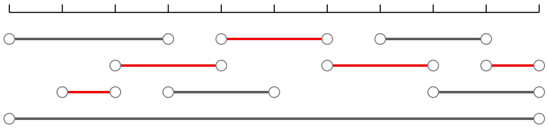
- ▷ the number of nodes (problems) in recursion tree is 2^n
- ▷ but the number of distinct problems is $n+1$

Remarks on DP

- DP is applicable for optimization problems with many overlapping subproblems
- DP is essentially brute force, which saves the time for repeatedly computing overlapping subproblems
- carefully use cut-and-paste argument (*cf.* s.8-9, week 7-2)
- unapplicable example: maximum subarray (*cf.* s.7, week 4-1)
  no overlapping subproblems

An Activity Selection Problem

- n activities use the same resource, *e.g.*, lecture hall, which serves one activity at a time
- each activity i has a start time s_i and finish time f_i
- goal: select a maximum number of compatible activities



- what is a greedy strategy?

  earliest start time first or earliest finish time first?

Earliest Finish Time First

ACTIVITY-EARLIEST-FINISH-TIME-FIRST(s, f)

```
sort activities w.r.t. finish time, i.e. f[1] ≤ f[2] ≤ ... ≤ f[n];
A = {1};
k = 1;                                    // activity with lastest finish time in A
for i = 2 to n do
    if s[i] ≥ f[k] then
        A = A ∪ {i};
        k = i;
return A;
```

Correctness

- assume greedy $= i_1, \ldots, i_s$ and OPT $= \{j_1, \ldots, j_t\}$ with $t > s$

$$f_{i_1} \le f_{i_2} \le \ldots \le f_{i_s}$$
$$f_{j_1} \le f_{j_2} \le \ldots \le f_{j_t}$$

- assume r is the first index s.t. $i_r \neq j_r$
- by greedy choice, we know $f_{i_r} \le f_{j_r}$
- replacing j_r by i_r in OPT yields another optimal solution

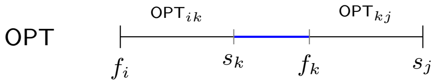
$$\text{OPT}_1 = \{i_1, \ldots, i_r, j_{r+1}, \ldots, j_t\},$$

- by induction, we can show that $i_l = j_l$ for $l = 1, \ldots, s$.
- greedy must also select $j_{s+1}, \ldots, j_t$, as they are compatible with $\{i_1, \ldots, i_s\}$, thereby implying $t = s$

Dynamic Programming

order activities according to finish time, *i.e.* $f_1 \leq \ldots \leq f_n$

consider OPT for instance S_{ij}: activities starting after f_i and finishing before $s_j (i < j)$



optimal substructure: OPT_{ik} and OPT_{kj} are optimal for S_{ik} and S_{kj} respectively

$$c[i,j] := \text{opt. objective} = c[i,k] + c[k,j] + 1$$

recursive formula

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

opt $= c[0, n+1]$, where $f_0 = 0$ and $s_{n+1} = f_n$

running time: $O(n^3)$

- worse than ACTIVITY-EARLIEST-FINISH-TIME-FIRST
- but better than brute force $O(2^n)$

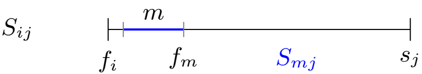
improvement: to reduce subproblems

we do not need to find k by brute force

Reducing Subproblems

Lemma

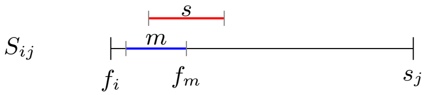
Consider any subproblem S_{ij} and there exists an optimal solution choosing an activity $m \in S_{ij}$ with minimum finish time.



$$c[i,j] = c[m,j] + 1 \text{ and opt} = c[0, n+1]$$

- $O(n)$ subproblems!
  running time $O(n)$, excluding the time for sorting
- let $S_k = \{i \mid s_i \geq f_k\}$: choose activity 1, then consider S_1; choose activity t, then consider S_t until some k with $S_k = \emptyset$

let $s \in$ OPT be the activity with minimum finish time



also assume that any OPT does not select m

replacing s by m in OPT yields another optimal solution OPT_1, a contradiction! need to check that OPT_1 is feasible!

Pseudocode

- assume $f_1 \leq f_2 \ldots \leq f_n$
- let $f_0 = 0$, thus $S_0 = \{1, \ldots, n\}$
- $\textsc{Recursive-Activity-Selector}(s, f, 0)$ returns OPT

$\textsc{Recursive-Activity-Selector}(s, f, k)$

$m = k + 1$;
while $m \leq n$ *and* $s[m] < f[k]$ **do**
    $m = m + 1$;   `// find` $m \in S_k$ `with min finish time`
if $m \leq n$ **then**
    return $\{m\} \cup \textsc{Recursive-Activity-Selector}(s, f, m)$;
else
    return $\emptyset$;   `//` $m > n$ `implies` $S_k = \emptyset$`, then terminates`

Greedy Paradigm

Direct greedy

1. develop a greedy strategy by intuition: always making locally optimal choices
2. prove the correctness by contradiction

Greedy via Dynamic Programming

1. determine the optimal substructure
2. develop a recursive solution
3. greedy choice yields only one subproblem
4. obtain a greedy algorithm