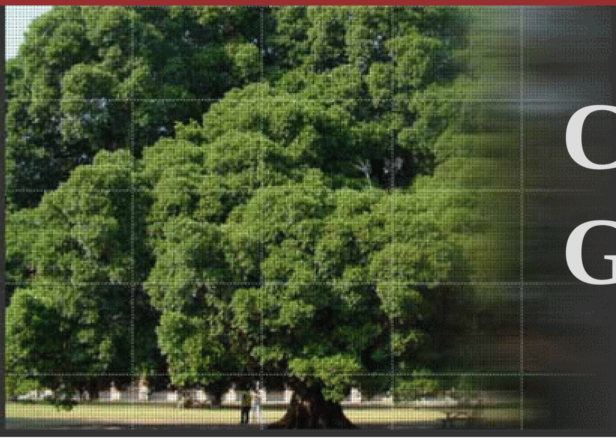
National Cheng Kung University



Chapter 16
Greedy Algorithms

Sun-Yuan Hsieh

謝孫源 教授

成功大學資訊工程學系

Introduction

- Similar to dynamic programming.

  Use for optimization problems.

- ***Idea*:**

  When we have a choice to make, make the one that looks best right now. Make a *locally optimal choice* in hope of getting a *globally optimal solution*

Activity selection

- n activities require *exclusive* use of a common resource. For example, scheduling the use of a classroom.

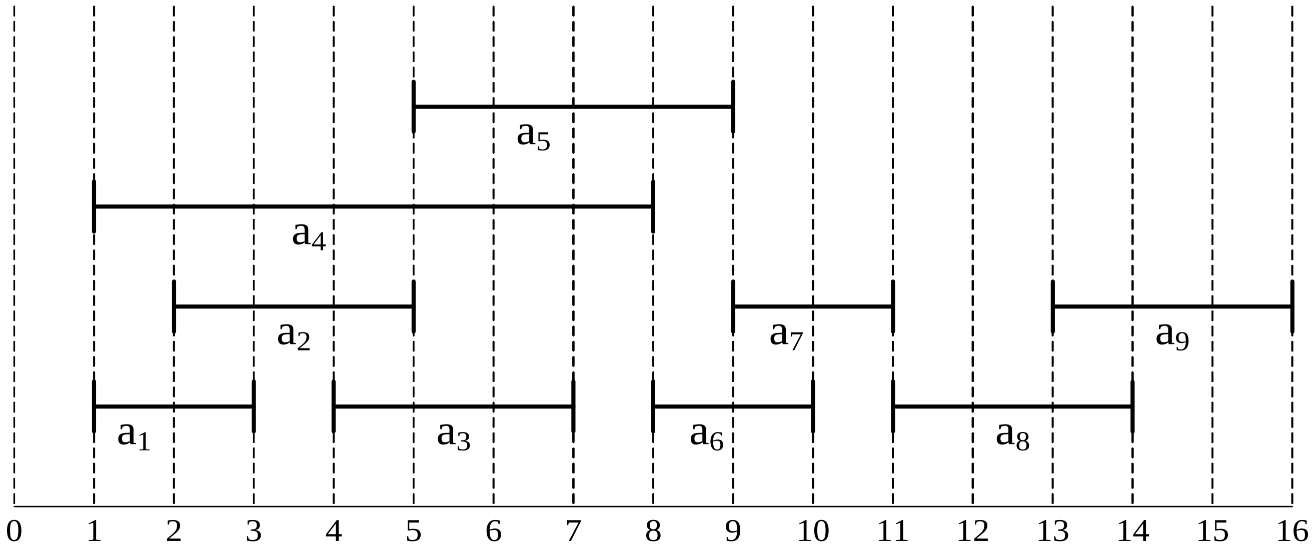
  Set of actives S={a_1,…, a_n}.

- a_i needs resource during period [s_i , f_i) , which is a half-open interval, where s_i = start time and f_i = finish time.

- ***Goal:*** Select the largest possible set of nonoverlapping (mutually compatible) activities

- ***Example*** ： S sorted by finish time:

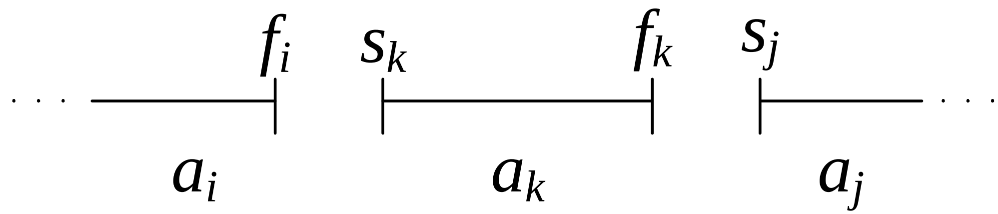
i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



- Maximum-size mutually compatible set:[a_1,a_3,a_6,a_8].
  Not unique: also[a_2,a_5,a_7,a_9].

Optimal substructure of activity selection

- $S_{ij} = \{a_k \in S : f_i \le s_k < f_k \le s_j\}$
  = activities that start after a_i finishes and finish before a_j starts.



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

To represent the entire problem, add fictitious activities:

- a_0=[-∞,0)
- a_{n+1}=[∞, "∞+1")

- We don't care about $-\infty$ in a_0 or "$\infty+1$" in a_n+1.

  Then $S = S_{0,n+1}$

  Range for S_{ij} is $0 \le i, j \le n+1$.
- Assume that activities are sorted by monotonically increasing finish time ：

  $f_0 \le f_1 \le f_2 \le \ldots \le f_n < f_{n+1}$

  Then $i \ge j \Rightarrow S_{ij} = \varnothing$.
  - If there exist $a_k \in S_{ij}$:

    $f_i \le s_k < f_k \le s_j < f_j \Rightarrow f_i < f_j$.
  - But $i \ge j \Rightarrow f_i \ge f_j$, Contradiction.

  So only need to worry about S_{ij} with $0 \le i < j \le n+1$.

  All other S_{ij} are $\varnothing$.

  Suppose that a solution to S_{ij} includes a_k. Have 2 subproblems:
  - S_{ik} (start after a_i finishes, finish before a_k starts)
  - S_{kj} (start after a_k finishes, finish before a_j starts)

- Let A_{ij} = optimal solution to S_{ij}.

  So $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, assuming:
  - S_{ij} is nonempty, and
  - We know a_k

Recursive solution to activity selection

$c[i,j]$ = size of maximum-size subset of mutually compatibles in S_{ij}.

- $i \geq j \Rightarrow$
  - If $S_{ij} = \varnothing \Rightarrow c[i, j] = 0$.
- $i < j \Rightarrow$

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

Theorem

Let $S_{ij} \neq \emptyset$, and let a_m be the activity in S_{ij} with the earliest finish time : $f_m = \min\ \{ f_k : a_k \in S_{ij} \}$. Then

1. a_m is used in some maximum-size subset of mutually compatible avtivities of S_{ij}.
2. $S_{im} = \emptyset$, so that choosing a_m leaves S_{mj} as only nonempty subproblem.

Proof

1. Let A_{ij} be a maximum-size subset of mutually compatible activities in S_{ij},
   Order activities in A_{ij} in monotonically increasing order of finish time.
   Let a_k be the first activity in A_{ij}.
   If a_k=a_m, done (a_m is used in a maximum-size subest).
   Otherwise, construct $A'_{ij} = A_{ij} - \{a_k\} \cup \{a_m\}$ (replace a_k by a_m).

Theorem

Let $S_{ij} \neq \emptyset$, and let a_m be the activity in S_{ij} with the earliest finish time :
$f_m = \min \{ f_k : a_k \in S_{ij} \}$. Then

1. a_m is used in some maximum-size subset of mutually compatible avtivities of S_{ij}.
2. $S_{im} = \emptyset$, so that choosing a_m leaves S_{mj} as only nonempty subproblem.

Proof

2. Suppose there is some $a_k \in S_{im}$. Then $f_i \leq s_k < f_k \leq s_m < f_m \Rightarrow f_k < f_m$.
   Then $a_k \in S_{ij}$ and it has an earlier finish time than f_m, which contradicts our choice of a_m.
   Therefore, there is no $a_k \in S_{im} \Rightarrow S_{im} = \emptyset$.

- ***Claim***

Activities in A'_{ij} are disjoint.

Proof

Activities in A_{ij} are disjoint, a_k is the first activity in A_{ij} to finish, $f_m \le f_k$ (so a_m doesn't overlap anything else in A'_{ij}). ◆(claim)

Since $|A'_{ij}| = |A_{ij}|$ and A_{ij} is a maximum-size subset, so is A'_{ij}. ◆(theorem)

This is great :

	before theorem	after theorem
# of subproblems in optimal solution	2	1
# of choices to consider	$j - i - 1$	1

- How we can solve top down:
- To solve a problem S_{ij}
  - Choose $a_m \in S_{ij}$ with earliest finish time: ***the greedy choice***
  - Then solve S_{mj}
- What are the subproblems?
  - Original problem is $S_{0,\, n+1}$
  - Suppose our first choice is a_{m1}
  - Then next subproblem is $S_{m1,\, n+1}$
  - Suppose next choice is a_{m2}
  - Nextsuproblem is $S_{m2,n+1}$
  - And so on

- ***Easy recursive algorithm:***
  Assumes activities already sorted by monotonically increasing finish time.
  (If not, then sort in $O(n\lg n)$ time)

  Return an optimal solution for $S_{i,n+1}$:

- REC-ACTIVITY-SELECTOR(s , f , i, n)
  $m \leftarrow i+1$
  while $m \leq n$ and $s_m < f_i$ ▶ Find first activity in $S_{i,n+1}$
  do $m \leftarrow m+1$
  if $m \leq n$
  then return $\{a_m\} \cup$REC-ACTIVITY-SELECTOR(s , f , m , n)
  else
  return 0

- ***Initial call:*** REC-ACTIVITY-SELECTOR(s , f , 0 , n)
- ***Time:*** $\Theta(n)$ — each activity examined exactly once.

Can make this iterative. It's already almost tail recursive.

- GREEDY-ACTIVITY-SELECTOR(s , f , n)

$A \leftarrow \{a_1\}$

$i \leftarrow 1$

for $m \leftarrow 2$ to n

do if $s_m \geq f_i$

then $A \leftarrow A \cup \{a_m\}$

$i \leftarrow m$ ► a_i is most recent addition to A

return A

Time: $\Theta(n)$.

- Greedy Strategy (typical streamline steps):
  1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
  2. Prove that there’s always an optimal solution that make the greedy choice, so that the greedy choice is always safe.
  3. Show that greedy choice and optimal solution to subprblem $\Rightarrow$ optimal solution to the problem.

- No general way to tell if a greedy algorithm is optimal, but two key ingredients are
  1. greedy-choice property and
  2. optimal substructure.

- **Greedy-choice property**

  A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.

- **Dynamic programming**
  - Make a choice at each step.
  - Choice depends on knowing optimal solutions to subproblems. Solve subproblems *first*.
  - Solve *bottom-up*.

- **Greedy**
  - Make a choice at each step.
  - Make the choice *before* solving the subproblems
  - Solve *top-down*.

- **Optimal substructure**

  Just show that optimal solution to subproblem and greedy choice $\Rightarrow$ optimal solution to problem.

- **Greedy vs. dynamic programming**

  The knapsack problem is a good example of the difference.

- ***0-1 knapsack problem***
  - n items.
  - Item i is worth $\$v_i$, weighs w_i pounds.
  - Find a most valuable subset of items with total weight $\leq W$.
  - 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 fraction of an item.
  - Both have optimal substructure.
  - But the fractional knapsack problem has the greedy-choice property, and 0-1 knapsack problem does not.
  - To solve the fractional problem, rank items by value/weight: v_i/w_i .

  Let $v_i/w_i \geq v_{i+1}/w_{i+1}$ for all i.

- FRACTIONAL-KNAPSACK(v , w , W)

  $load \leftarrow 0$

  $i \leftarrow 1$

  while $load < W$ and $i \leq n$

      do if $w_i \leq W\text{-}load$

          then take all of item i

          else take $(W\text{-}load)/w_i$ of item i

      add what was taken to *load*

      $i \leftarrow i+1$

  Time: O($n\lg n$) to sort, O(n) thereafter.

Greedy don't work for 0-1 knapsack problem

$W = 50$

- Greedy solution :
  - Take items 1 and 2.
  - value = 160, weight = 30.

Have 20 pounds of capacity left over.

- Optimal solution :
  - Take items 2 and 3.
  - value = 220, weight = 50.
  - No leftover capacity.

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