Greedy Algorithms

Reference:
CLRS Chapter 16

Topics:
• Activity-selection problem
• Elements of the greedy strategy
• Huffman coding

Greedy Method

• Similar to dynamic programming
• Used 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 to getting a globally optimal solution.

• Greedy algorithms don’t always yield an optimal solution. But sometimes they do. We’ll see a problem for which they do. Then we’ll look at some general characteristics of when greedy algorithms give optimal solutions.

Activity-Selection Problem

• $n$ activities requires exclusive use of a common resource. For example, scheduling the use of a classroom.
• Set of activities $S = \{a_1, a_2, \ldots, a_n\}$.
• $a_i$ 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.

• **Note:** Could have many other objectives:
  – Schedule room for longest time.
  – Maximize income rental fees.

Activity-Selection Problem

• Example: $S$ sorted by finish time:

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>$f_i$</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

• 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

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

  \[ \cdots a_i \cdots \langle f_i \rangle \quad a_k \quad \langle f_k \rangle \quad a_j \quad \cdots \]

- 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:
  - $a_0 = (-\infty, 0)$,
  - $a_{n+1} = (\infty, \infty)$

- We don't care about $-\infty$ in $a_0$ or $\infty$ in $a_{n+1}$.
- Then $S = S_{i, n+1}$ for $0 \leq i, j \leq n+1$.

Solution to $S_{ij}$ is (solution to $S_{ik} \cup \{a_k\} \cup$ (solution to $S_{kj}$).
Since $a_k$ is in neither subproblem, and the subproblems are disjoint,
|Solution to $S_{ij}$| = |solution to $S_{ik}$| + 1 + |solution to $S_{kj}$|

If an optimal solution to $S_{ij}$ includes $a_k$, then the solutions to $S_{ik}$ and $S_{kj}$ used within this solution must be optimal as well. Use the usual cut-and-paste argument.

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

- $c[i, j] = \text{size of maximum-size subset of mutually compatible activities in } S_{ij}$.
  - $i \geq j \Rightarrow S_{ij} = \emptyset \Rightarrow c[i, j] = 0$.
  - $S_{ij} \neq \emptyset$, suppose we know that $a_k$ is in the subset. Then
    \[ c[i, j] = c[i, k] + c[k, j] + 1 \]
  - But of course we don’t know which $k$ to use, and so
    \[ c[i, j] = \begin{cases} 0, & \text{if } S_{ij} = \emptyset \\ \max\{c[i, k] + c[k, j] + 1\}, & \text{if } S_{ij} \neq \emptyset \end{cases} \]

- Why this range of $k$? Because $\{a_k \in S : f_i \leq s_k < f_j \leq s_j \} \Rightarrow a_k$ can’t be $a_i$ or $a_j$. Also need to ensure that $a_k$ is actually in $S_{ij}$. Since $i < k < j$ is not sufficient on its own to ensure this.

- Assume activities are sorted by increasing finish time: $f_i \leq f_1 \leq f_2 \leq \ldots \leq f_n < f_{n+1}$
  - If there exists $a_k \in S_{ij}$, $f_i \leq s_k < f_j \leq s_j \Rightarrow f_i < f_j$.
  - But $i \geq j \Rightarrow f_i < f_j$, a contradiction.

- So only need to worry about $S_{ij}$ with $0 \leq i < j \leq n+1$.
All other $S_{ij}$ are $\emptyset$.

- 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)
Theorem

- Let $S_y \neq \emptyset$ and let $a_m$ be the activity in $S_y$ with the earliest finish time: $f_m = \min \{ f_k : a_k \in S_y \}$. Then,
  1. $a_m$ is used in some maximum-size subset of mutually compatible activities of $S_y$.
  2. $S_m = \emptyset$, so that choosing $a_m$ leaves $S_{mj}$ as the only one nonempty subproblem.

Proof.

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, $f_m \leq f_k$ (so $a_m$ doesn't overlap anything else in $A'_{ij}$). \( \square \) (claim)

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

Recursive Solution to Activity Selection

- 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 $a_{m2}$.
  - Next subproblem is $S_{m2,n+1}$.
  - And so on.

- Each subproblem is $S_{mi,n+1}$, i.e., the last activities to finish. And the subproblems chosen have finish times that increase. Therefore, we can consider each activity just once, in monotonically increasing order of finish time.

Recursive Solution to Activity Selection

- This is great:
  - # of subproblems in optimal solution
  - # of choices to consider

<table>
<thead>
<tr>
<th>before theorem</th>
<th>after theorem</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Now we can solve top down:
- Choose $a_m \in S_y$ 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 $a_{m2}$.
- Next subproblem 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 $O(n \log n)$ time.) Return an optimal solution for $S_{i,n+1}$.

```
RECURSIVE ACTIVITY SELECTION
1 m ← i + 1
2 while m ≤ n and s_m < f_i  // Find the first activity in S_y.
3     do m ← m + 1
4 if m ≤ n
5     then return{a_m} ∪ RECURSIVE ACTIVITY SELECTION(s, f, m, n)
6 else return φ
```

- Initial call: REC-ACTIVITY-SELECTON(s, f, 0, n).
Easy Recursive Algorithm

• Idea: The while loop checks $a_{i+1}, a_{i+2}, \ldots, a_n$ until it finds an activity $a_m$ that is compatible with $a_i$ (need $s_m \geq f_i$).
  – If the loop terminates because $a_m$ is found ($m \leq n$), then recursively solve $S_{m,n+1}$, and return this solution, along with $a_m$.
  – If the loop never finds a compatible $a_m$ ($m > n$), then just return empty set.

Go through example given earlier. Should get $\{a_1, a_3, a_6, a_8\}$

• Time: $\Theta(n)$ — each activity examined exactly once.

• Can make this iterative. It’s already almost tail recursive.

Go through example given earlier. Should again get $\{a_1, a_3, a_6, a_8\}$

• Time: $\Theta(n)$.

ITERATIVE ACTIVITY SELECTION

ITERATIVE-ACTIVITY-SELECTION(s,f,n)
1 A ←\{a_1\}
2 i ← 1
3 for m ← 2 to n
4     do if $s_m \geq f_i$
5         then A ← A ∪\{a_m\}
6             i ← m //a_i is most recent addition to A
7 return A

Greedy Strategy

• The choice that seems best at the moment is the one we go with.
  What did we do for activity selection?
  – Determine the optimal substructure.
  – Develop a recursive solution.
  – Prove that at any stage of recursion, one of the optimal choices is the greedy choice. Therefore, it’s always safe to make the greedy choice.
  – Show that all but one of the subproblems resulting from the greedy choice are empty.
  – Develop a recursive greedy algorithm.
  – Convert it to an iterative algorithm.

• At first, it looks like dynamic programming. Typically, we streamline these steps.

Greedy Strategy

• Develop the substructure with an eye toward
  – Making the greedy choice.
  – Leaving just one subproblem.

• For activity selection, we showed that the greedy choice implied that in $S_{ip}$ only $i$ varied, and $j$ was fixed at $n+1$.

• We could have started out with a greedy algorithm in mind.
  – Define $S_i = \{a_k \in S : f_j \leq s_k\}$.
  – Then show that the greedy choice – first $a_m$ to finish in $S_i$ – combined with optimal solution to $S_m$ ⇒ optimal solution to $S_i$. 
Greedy Strategy

- Typical streamlined steps:
  - 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.
  - Show that greedy choice and optimal solution to subproblem ⇒ optimal solution to the problem.

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

Greedy Strategy

- 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.

Greedy Strategy

- Typically show the greedy-choice property by what we did for activity selection:
  - Look at a globally optimal solution.
  - If it includes the greedy choice, done.
  - Else, modify it to include the greedy choice, yielding another solution that's just good.

- Can get efficiency gains from greedy-choice property.
  - Preprocess input to put it into greedy order.
  - Or, if dynamic data, use a priority queue.

- Optimal substructure
  - Just show that optimal solution to subproblem and greedy choice ⇒ optimal solution to the problem.