COMP251: Greedy algorithms

Jérôme Waldspühl
School of Computer Science
McGill University

Based on (Cormen et al., 2002)
Based on slides from D. Plaisted (UNC) & (goodrich & Tamassia, 2009)

Overview

• Algorithm design technique to solve optimization problems.
• Problems exhibit optimal substructure.
• Idea (the greedy choice):
  – 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.

Greedy Strategy

The choice that seems best at the moment is the one we go with.
– Prove that when there is a choice to make, one of the optimal choices is the greedy choice. Therefore, it is always safe to make the greedy choice.
– Show that all but one of the sub-problems resulting from the greedy choice are empty.

Activity-selection Problem

• Input: Set $S$ of $n$ activities, $a_1, a_2, ..., a_n$.
  – $s_i$ = start time of activity $i$.
  – $f_i$ = finish time of activity $i$.
• Output: Subset $A$ of maximum number of compatible activities.
  – 2 activities are compatible, if their intervals do not overlap.

Example:

<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>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>$f_i$</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Activities sorted by finishing time.

Optimal compatible set: $\{a_1, a_4, a_5\}$

Optimal Substructure

• Assume activities are sorted by finishing times.
• Suppose an optimal solution includes activity $a_i$. This solution is obtained from:
  – An optimal selection of $a_1, ..., a_{i-1}$ activities compatible with one another, and that finish before $a_i$ starts.
  – An optimal solution of $a_{i+1}, ..., a_n$ activities compatible with one another, and that start after $a_i$ finishes.
**Optimal Substructure**

- Let \( S_i \) = subset of activities in \( S \) that start after \( a_i \) finishes and finish before \( a_i \) starts.
  \( S_i = \{ a_j \in S : \forall i, j \quad f_i \leq s_j < f_j \leq s_j \} \)
- \( A_i \) = optimal solution to \( S_i \)
- \( A_i \) = \( A_{i+1} \cup \{ a_i \} \cup A_i \)

**Recursive Solution**

- Subproblems: Selecting maximum number of mutually compatible activities from \( S_i \)
- Let \( c[i,j] \) = size of maximum-size subset of mutually compatible activities in \( S_i \)

Recursive solution:

\[
c[i,j] = \begin{cases} 
0 & \text{if } S_i = \emptyset \\
\max_{i \leq k < j} \{ c[i,k] + c[k+1,j] + 1 \} & \text{if } S_i \neq \emptyset
\end{cases}
\]

Note: We do not know which \( k \) to use for the optimal solution.

---

**Greedy choice**

**Theorem:**
Let \( S_i \neq \emptyset \), and let \( a_m \) be the activity in \( S_i \) with the earliest finish time: \( f_m = \min\{ f_i : a_i \in S_i \} \). Then:

1. \( a_m \) is used in some maximum-size subset of mutually compatible activities of \( S_i \).
2. \( S_{ij} \neq \emptyset \), so that choosing \( a_m \) leaves \( S_{im} \) as the only nonempty subproblem.

**Proof:**

(2nd part) If there is \( a_n \in S_n \) then \( f_s, s_i < f_n < s_n \) and \( f_s < f_i \), which contradicts the hypothesis that \( a_m \) has the earlier finish.

(1st part)
- Let \( A_i \) be a maximum-size subset of mutually compatible activities in \( S_i \); i.e., an optimal solution of \( S_i \).
- Order activities in \( A_i \) in monotonically increasing order of finish time, and let \( a_i \) be the activity with earliest finish time.
- If \( a_i = a_m \) then done.
- Otherwise, construct \( A' \rightarrow A_i \cup \{ a_m \} \)
- By definition, \( a_m \) finishes before \( a_i \). Thus, \( A' \) does not overlap with anything in \( A_i \).
- \( |A_j| = |A' \rangle \) and \( A_i \) maximal, thus \( A' \rightarrow A_i \).

---

**Greedy choice**

<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>
</tr>
</thead>
<tbody>
<tr>
<td>( s_i )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>( f_i )</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Activities sorted by finishing time.

---

**Activity-selection Problem**

We can now solve the problem top-down:
- Choose \( a_m \in S_i \) with the earliest finish time (greedy choice).
- Solve \( S_i \).
15-01-28

### Activity-selection Problem

<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>
</tr>
</thead>
<tbody>
<tr>
<td>s_i</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>f_i</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Activities sorted by finishing time.

### Recursive Algorithm

**Recursive-Activity-Selector** $(s, f, i, j)$

1. $m ← i+1$
2. while $m < j$ and $s_m < f_i$
3. do $m ← m+1$
4. if $m < j$
5. then return $(a_m) U$
6. else return φ

Initial Call: Recursive-Activity-Selector $(s, f, 0, n+1)$

Complexity: $Θ(n)$

Remark: Straightforward to convert the algorithm to an iterative one.

### Typical 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 imply optimal solution to the problem.
- Make the greedy choice and solve top-down.
- May have to preprocess input to put it into greedy order (e.g. sorting activities by finish time).

### Elements of Greedy Algorithms

No general way to tell if a greedy algorithm is optimal, but two key ingredients are:

- Greedy-choice Property.
  - A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
- Optimal Substructure.
Text Compression

• Given a string X, efficiently encode X into a smaller string Y
  – Saves memory and/or bandwidth
• A good approach: Huffman encoding
  – Compute frequency f(c) for each character c.
  – Encode high-frequency characters with short code words
  – No code word is a prefix for another code
  – Use an optimal encoding tree to determine the code words

Encoding Tree Example

• A code is a mapping of each character of an alphabet to a binary code-word
• A prefix code is a binary code such that no code-word is the prefix of another code-word
• An encoding tree represents a prefix code
  – Each external node stores a character
  – The code word of a character is given by the path from the root to the external node storing the character (0 for a left child and 1 for a right child)

Encoding Tree Optimization

• Given a text string X, we want to find a prefix code for the characters of X that yields a small encoding for X
  – Frequent characters should have long code-words
  – Rare characters should have short code-words
• Example
  – X = abracadabra
  – T₁ encodes X into 29 bits
  – T₂ encodes X into 24 bits

Example

X = abracadabra
Frequencies
<table>
<thead>
<tr>
<th>Character</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>5</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>2</td>
</tr>
</tbody>
</table>

Extended Huffman Tree Example

Extended Huffman’s Algorithm

• Given a string X, Huffman’s algorithm constructs a prefix code the minimizes the size of the encoding of X
• It runs in time $O(n + d \log d)$, where n is the size of X and d is the number of distinct characters of X
• A heap-based priority queue is used as an auxiliary structure

Huffman’s Algorithm

Algorithm HuffmanEncoding(X)
Input: string X of size n
Output: optimal encoding trie for X
\[ C \leftarrow \text{distinctCharacters}(X) \]
\[ \text{computeFrequencies}(C, X) \]
\[ Q \leftarrow \text{new empty heap} \]
for all $c \in C$
\[ T_c \leftarrow \text{new single-node tree storing } c \]
\[ Q.insert(getFrequency(c), T_c) \]
while Q.size() > 1
\[ f_1 \leftarrow Q.minKey() \]
\[ T_1 \leftarrow Q.removeMin() \]
\[ f_2 \leftarrow Q.minKey() \]
\[ T_2 \leftarrow Q.removeMin() \]
\[ T \leftarrow \text{join}(T_1, T_2) \]
\[ Q.insert(f_1 + f_2, T) \]
return Q.removeMin()