COMP251: Greedy algorithms

Jérôme Waldispühl
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, $a_1, a_2, ..., a_n$.
  – $s_i = \text{start time of activity } i$.
  – $f_i = \text{finish time of activity } i$.

• **Output:** Subset $A$ of maximum number of compatible activities.
  – 2 activities are compatible, if their intervals do not overlap.

Example:

Activities in each line are compatible.
## Activity-selection Problem

### 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.

### Diagram

Optimal compatible set: \{ $a_1$, $a_3$, $a_5$ \}
Optimal Substructure

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

- Let $S_{ij} = \text{subset of activities in } S \text{ that start after } a_i \text{ finishes and finish before } a_j \text{ starts.}$
  \[
  S_{ij} = \left\{ a_k \in S : \forall i, j \quad f_i \leq s_k < f_k \leq s_j \right\}
  \]

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

- $A_{ij} = A_{ik} \cup \{ a_k \} \cup A_{kj}$
Recursive Solution

- Subproblems: Selecting maximum number of mutually compatible activities from $S_{ij}$.
- Let $c[i, j] = \text{size of maximum-size subset of mutually compatible activities in } S_{ij}$.

Recursive solution:

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

Note: We do not know which $k$ to use for the optimal solution.
Greedy choice

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 activities of $S_{ij}$.
2. $S_{im} = \emptyset$, so that choosing $a_m$ leaves $S_{mj}$ as the only nonempty subproblem.
Greedy choice

Proof:
(2\textsuperscript{nd} part) If there is $a_k \in S_{im}$ then $f_i \leq s_k < f_k \leq s_m$ and $f_k < f_m$ which contradicts the hypothesis that $a_m$ has the earliest finish.

(1\textsuperscript{st} part)
• Let $A_{ij}$ be a maximum-size subset of mutually compatible activities in $S_{ij}$ (i.e. an optimal solution of $S_{ij}$).
• Order activities in $A_{ij}$ in monotonically increasing order of finish time, and let $a_k$ be the activity with earliest finish time.
• If $a_k = a_m$ then done.
• Otherwise, construct $A'_{ij} = A_{ij} - \{ a_k \} \cup \{ a_m \}$
• By definition, $a_m$ finish before $a_k$. Thus, $a_m$ does not overlap with anything in $A'_{ij}$.
• $|A_{ij}| = |A'_{ij}|$ and $A_{ij}$ maximal, thus $A_{ij} = A'_{ij}$. 
Greedy choice

<table>
<thead>
<tr>
<th></th>
<th>Before theorem</th>
<th>After theorem</th>
</tr>
</thead>
<tbody>
<tr>
<td># subproblems in optimal solution</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td># choices to consider</td>
<td>j-i-1</td>
<td>1</td>
</tr>
</tbody>
</table>

We can now solve the problem top-down:

- Choose $a_m \in S_{ij}$ with the earliest finish time (greedy choice).
- Solve $S_{ij}$. 
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.
### 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.
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.
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. \(m \leftarrow i+1\)
2. \(\textbf{while } m < j \text{ and } s_m < f_i\)
3. \(\textbf{do } m \leftarrow m+1\)
4. \(\textbf{if } m < j\)
5. \(\textbf{then return } \{a_m\} \cup \) \(\) \(\text{Recursive-Activity-Selector}(s, f, m, j)\)
6. \(\text{else return } \emptyset\)

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

Complexity: \(\Theta(n)\)

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 ⇒ 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
  – $T_1$ encodes $X$ into 29 bits
  – $T_2$ encodes $X$ into 24 bits

$T_1$

```plaintext
T1

c

a r

d b
```

$T_2$

```plaintext
T2

a

b r

c d
```
Example

\[ X = \text{abracadabra} \]

Frequencies

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Diagram of tree structure.
Extended Huffman Tree Example

String: a fast runner need never be afraid of the dark

<table>
<thead>
<tr>
<th>Character</th>
<th>a</th>
<th>b</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>h</th>
<th>i</th>
<th>k</th>
<th>n</th>
<th>o</th>
<th>r</th>
<th>s</th>
<th>t</th>
<th>u</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>9</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Huffman’s Algorithm

- Given a string $X$, Huffman’s algorithm construct a prefix code that 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.

```
Algorithm HuffmanEncoding(X)
Input string $X$ of size $n$
Output optimal encoding trie for $X$

$C \leftarrow$ distinctCharacters($X$)
computeFrequencies($C$, $X$)
$Q \leftarrow$ new empty heap
for all $c \in C$
    $T \leftarrow$ new single-node tree storing $c$
    $Q$.insert(getFrequency($c$), $T$)
while $Q$.size() > 1
    $f_1 \leftarrow Q$.minKey()
    $T_1 \leftarrow Q$.removeMin()
    $f_2 \leftarrow Q$.minKey()
    $T_2 \leftarrow Q$.removeMin()
    $T \leftarrow$ join($T_1$, $T_2$)
    $Q$.insert($f_1 + f_2$, $T$)
return $Q$.removeMin()
```
\begin{center}
\includegraphics[width=\textwidth]{diagram.png}
\end{center}