The Greedy Method and Compression

The Greedy Method Technique (§ 11.4.2)

**The greedy method** is a general algorithm design paradigm, built on the following elements:
- **configurations**: different choices, collections, or values to find
- **objective function**: a score assigned to configurations, which we want to either maximize or minimize

It works best when applied to problems with the **greedy-choice** property:
- A globally optimal solution can always be found by a series of local improvements from a starting configuration.

Text Compression (§ 11.4)

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**

- $X = \text{abracadabra}$
- $T_1$ encodes $X$ into $29$ bits.
- $T_2$ encodes $X$ into $24$ bits.

Huffman’s Algorithm

- Given a string $X$, Huffman’s algorithm constructs 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($X$)**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>string $X$ of size $n$</td>
<td>optimal encoding trie for $X$</td>
</tr>
</tbody>
</table>

1. $C \leftarrow \text{distinctCharacters}(X)$
2. $\text{computeFrequencies}(C, X)$
3. $Q \leftarrow \text{new empty heap}$
4. for all $c \in C$ do
   - $T \leftarrow \text{new single-node tree storing } c$
   - $Q.insert(getFrequency(c), T)$
5. while $Q.size() > 1$ do
   - $f_1 \leftarrow Q.minKey()$
   - $T_1 \leftarrow Q.removeMin()$
   - $f_2 \leftarrow Q.minKey()$
   - $T_2 \leftarrow Q.removeMin()$
   - $T \leftarrow \text{join}(T_1, T_2)$
   - $Q.insert(f_1 + f_2, T)$
6. return $Q.removeMin()$
The Fractional Knapsack Problem (not in book)

- Given: A set $S$ of $n$ items, with each item $i$ having:
  - $b_i$ - a positive benefit
  - $w_i$ - a positive weight

- Goal: Choose items with maximum total benefit but with weight at most $W$.

- If we are allowed to take fractional amounts, then this is the **fractional knapsack problem**.
  - In this case, we let $x_i$ denote the amount we take of item $i$
  - Objective: maximize $\sum_{i \in S} b_i (x_i/w_i)$
  - Constraint: $\sum_{i \in S} x_i \leq W$


Example

- Given: A set $S$ of $n$ items, with each item $i$ having:
  - $b_i$ - a positive benefit
  - $w_i$ - a positive weight

- Goal: Choose items with maximum total benefit but with weight at most $W$.

Example:

<table>
<thead>
<tr>
<th>Items:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight:</td>
<td>4 ml</td>
<td>8 ml</td>
<td>2 ml</td>
<td>6 ml</td>
<td>1 ml</td>
</tr>
<tr>
<td>Benefit:</td>
<td>$12$</td>
<td>$32$</td>
<td>$40$</td>
<td>$30$</td>
<td>$50$</td>
</tr>
<tr>
<td>Value: ($per ml$)</td>
<td>3</td>
<td>4</td>
<td>20</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>

Solution:

- 1 ml of 5
- 2 ml of 3
- 6 ml of 4
- 1 ml of 2

Task Scheduling (not in book)

- Given: a set $T$ of $n$ tasks, each having:
  - A start time, $s_i$
  - A finish time, $f_i$ (where $s_i < f_i$)

- Goal: Perform all the tasks using a minimum number of "machines."

Task Scheduling:

<table>
<thead>
<tr>
<th>Machine 1</th>
<th>Machine 2</th>
<th>Machine 3</th>
<th>Machine 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Greedy Method and Compression
Task Scheduling Algorithm

- Greedy choice: consider tasks by their start time and use as few machines as possible with this order.
- Run time: O(n log n). Why?
- Correctness: Suppose there is a better schedule.
  - We can use k-1 machines
  - The algorithm uses k
  - Let i be first task scheduled on machine k
  - Machine i must conflict with k-1 other tasks
  - But that means there is no non-conflicting schedule using k-1 machines

Algorithm `taskSchedule(T)`
\[ m \leftarrow 0 \]
while \( T \) is not empty
\[ m \leftarrow m + 1 \]
remove task i w/ smallest \( s_i \)
if there’s a machine j for i then
  schedule i on machine j
else
  schedule i on machine m

Example

- Given: a set \( T \) of n tasks, each having:
  - A start time, \( s_i \)
  - A finish time, \( f_i \) (where \( s_i < f_i \))
  - \([1,4], [1,3], [2,5], [3,7], [4,7], [6,9], [7,8]\) (ordered by start)
- Goal: Perform all tasks on min. number of machines

The tasks are scheduled as follows:

<table>
<thead>
<tr>
<th>Machine 1</th>
<th>Machine 2</th>
<th>Machine 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,4]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[1,3]</td>
<td>[3,7]</td>
<td></td>
</tr>
<tr>
<td>[2,5]</td>
<td>[4,7]</td>
<td></td>
</tr>
<tr>
<td>[6,9]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[7,8]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© 2004 Goodrich, Tamassia