Lecture: Analysis of Algorithms (CS483 - 001)

Amarda Shehu

Spring 2017
1 Greedy Algorithms
   • In the Context of the Following Problems
     • The Fractional Knapsack Problem
     • Huffman Coding

2 Summary
Sample Problems to Illustrate Greedy Algorithms

- The Fractional Knapsack Problem
- Variable-length (Huffman) Coding
The Fractional Knapsack Problem

- Given $n$ objects
- Each object has an integer profit $p_i$
- Each object has a fractional weight $w_i$
- You can take fractions of an object
- You have a knapsack with weight capacity $M$, where $M$ is not necessarily an integer
- Problem: Fit objects (taking even fractions of them) that give the maximum total profit
An Optimal Greedy Solution to the Fractional Knapsack Problem

- Sort the items by descending $p_i/w_i$ ratios (focusing on maximizing profit while minimizing weight)
- Examine each object $i \in \{1, \ldots, n\}$ in this order
- If object fits in knapsack, take it
- What is the time complexity?
- Why does this greedy approach find the optimal solution to the Fractional Knapsack Problem?
Proof of Correctness

Let $X \in \{1, 2, \ldots, k\}$ be the optimal items taken

- Consider item $j$ with associated $(p_j, w_j)$ that has the highest $p_j/w_j$ ratio
- If $j$ is not used in $X$, then $X$ is not optimal: We can remove portions of items with a total weight of $w_j$ from $X$ and add $j$ instead.
- Repeating this process, you see that the greedy approach changes $X$ considering all items without decreasing the total value of $X$. 
Another Problem Addressed with a Greedy Algorithm

Huffman (Variable) Coding
The Coding Problem

Consider a message consisting of \( k \) characters (with known frequencies).

We want to encode this message using a binary cipher.

That is, we want to assign \( d \) bits to each letter:

<table>
<thead>
<tr>
<th>Letter</th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
<th>( d )</th>
<th>( e )</th>
<th>( f )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency ( \times 10^3 )</td>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>Fixed-length encoding</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
</tr>
</tbody>
</table>

A message consisting of 100,000 \( a-f \) characters would require 300,000 bits of storage!!!
How about Variable-length Encoding?

- We could assign a variable-length encoding instead:

<table>
<thead>
<tr>
<th>Letter</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency ($\times 10^3$)</td>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>Fixed-length encoding</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
</tr>
<tr>
<td>Variable-length encoding</td>
<td>0</td>
<td>101</td>
<td>100</td>
<td>111</td>
<td>1101</td>
<td>1100</td>
</tr>
</tbody>
</table>

- A message like 001011101 parses uniquely
  - That is to say that one can decode this cipher uniquely
  - This result is based on the fact that no code is a prefix of another for the encoded characters

- Only 9 bits are used instead.
Problem: Given an alphabet $A = \{a_1, \ldots, a_n\}$ with frequency distribution $f(a_i)$, find a binary prefix code $C$ for $A$ that minimizes the number of bits

$$B(C) = \sum_{i=1}^{n} f(a_i) \cdot L(c(a_i))$$

needed to encode a message of $\sum_{i=1}^{n} f(a_i)$ characters, where $c(a_i)$ is the codeword/code for encoding $a_i$, and $L(c(a_i))$ is the length of this code.

Solution: Huffman developed a greedy algorithm for producing a minimum-cost prefix code. The code that is produced is called a Huffman Code.
Basic Idea Behind Huffman Coding

- A binary tree constructs codes
- 1-1 correspondence between the leaves and the characters
- The label of each leaf is the frequency of each character
- Left edges are labeled 0, right edges are labeled 1
- Path from root to leaf is the code associated with the character at that leaf

{a = 000, b = 001, c = 010, d = 011, e = 1}
Basic Idea Behind Huffman Coding

**Step 1.** Pick two letters $x, y$ from alphabet $A$ with the smallest frequencies and create a subtree that has these two characters as leaves. This is the greedy idea. Label the root of this subtree as $z$.

**Step 2.** Set frequency $f(z) = f(x) + f(y)$. Remove $x$ and $y$ and add $z$, creating a new alphabet $A' = A \cup z - \{x, y\}$. Note that $|A'| = |A| - 1$

Repeat this procedure, called *merge*, creating new alphabet $A'$ until only one symbol is left. The resulting tree is the **Huffman Code**.
Huffman Code Algorithm

HuffmanCoding(C)
1: \( n \leftarrow |A| \)
2: \( Q \leftarrow A \)
3: for all \( i = 1 \) to \( n - 1 \) do
4: allocate a new node \( z \)
5: left[\( z \)] \( \leftarrow x \leftarrow \) EXTRACT-MIN(\( Q \))
6: right[\( z \)] \( \leftarrow y \leftarrow \) EXTRACT-MIN(\( Q \))
7: \( f[z] \leftarrow f[x] + f[y] \)
8: INSERT(\( Q, z \))
9: return \( \) EXTRACT-MIN(\( Q \))

Can you see why the time complexity of this algorithm is \( O(n \cdot \log n) \)?
Greedy Algorithms (for Optimization Problem)

- A greedy algorithm builds a solution one step at a time.
- Unlike DP, at each step, a greedy algorithm makes the currently best choice from a small number of choices.
- The currently best choice is also referred to as the locally optimal choice.
- Greedy algorithms are similar to DP algorithms in:
  - the solution is efficient if the problem exhibits substructure.
- BUT
  - The greedy solution may not be optimal even if the problem exhibits optimal substructure (DP needed in such cases).
When to Design Greedy Algorithms

- Greedy-choice property: a “locally optimal” choice leads to a “globally optimal” solution
- Applying the greedy approach to other problems that do not have this property can yield suboptimal solutions
- BUT: suboptimal solutions may be good enough approximations of the optimal solution on some applications
  - Instance: when globally optimal solution is too expensive to compute