Greedy Algorithms

Introduction
Activity Selection
Knapsack
Huffman Coding

Greedy Programming Definition ........................................ 2
Activity Selection .......................................................... 3
Knapsack Problem ......................................................... 4
Huffman Coding ............................................................ 5
Huffman Trees ............................................................... 6
Huffman Algorithm ......................................................... 7
Huffman Illustration ....................................................... 8

Greedy Programming Definition

Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step. For many optimization problems, using dynamic programming to determine the best choices is overkill; simpler, more efficient algorithms will do. A greedy algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution (or close to globally optimal).

Activity Selection

Suppose we have a set $S = \{a_1, a_2, \ldots, a_n\}$ of $n$ proposed activities that wish to use a resource, such as a lecture hall, which can serve only one activity at a time. Each activity $a_i$ has a start time $s_i$ and a finish time $f_i$, where $0 \leq s_i < f_i < \infty$. If selected, activity $a_i$ takes place during the half-open time interval $[s_i, f_i)$. Activities $a_i$ and $a_j$ are compatible if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. That is, $a_i$ and $a_j$ are compatible if $s_i \geq f_j$ or $s_j \geq f_i$. In the activity-selection problem, we wish to select a maximum-size subset of mutually compatible activities. We assume that the activities are sorted in monotonically increasing order of finish time:

$$f_1 \leq f_2 \leq f_3 \leq \cdots \leq f_{n-1} \leq f_n.$$  \hfill (16.1)

(We shall see later the advantage that this assumption provides.) For example, consider the following set $S$ of activities:

<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>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>$f_i$</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>11</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>
Knapsack Problem

The 0-1 knapsack problem is the following. A thief robbing a store finds \( n \) items. The \( i \)th item is worth \( v_i \) dollars and weighs \( w_i \) pounds, where \( v_i \) and \( w_i \) are integers. The thief wants to take as valuable a load as possible, but he can carry at most \( W \) pounds in his knapsack, for some integer \( W \). Which items should he take? (We call this the 0-1 knapsack problem because for each item, the thief must either take it or leave it behind; he cannot take a fractional amount of an item or take an item more than once.)

In the fractional knapsack problem, the setup is the same, but the thief can take fractions of items, rather than having to make a binary (0-1) choice for each item.

---

Huffman Coding

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

Frequency (in thousands)

Fixed-length codeword

Variable-length codeword

000 001 010 011 100 101

0 101 100 111 1101 1100

we consider the problem of designing a binary character code (or code for short) in which each character is represented by a unique binary string, which we call a codeword. If we use a fixed-length code, we need 3 bits to represent 6 characters: \( a = 000 \), \( b = 001 \), \( c = 010 \), \( d = 011 \), \( e = 100 \), \( f = 101 \). This method requires 300,000 bits to code the entire file. Can we do better?

A variable-length code can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long codewords. Figure 16.3 shows such a code; here the 1-bit string 0 represents \( a \), and the 4-bit string 1100 represents \( f \). This code requires

\[
(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1,000 = 224,000 \text{ bits}
\]

---

Huffman Trees

(a)

(b)

---

Huffman Algorithm

\[
\text{HUFFMAN}(C)\\n1 \quad n = |C|\\n2 \quad Q = C\\n3 \quad \text{for } i = 1 \text{ to } n - 1\\n4 \quad \text{allocate a new node } z\\n5 \quad z.left = x = \text{EXTRACT-MIN}(Q)\\n6 \quad z.right = y = \text{EXTRACT-MIN}(Q)\\n7 \quad z.freq = x.freq + y.freq\\n8 \quad \text{INSERT}(Q, z)\\n9 \quad \text{return } \text{EXTRACT-MIN}(Q) // \text{return the root of the tree}
\]
Huffman Illustration

(a) \(f:5 \quad e:9 \quad c:12 \quad b:13 \quad d:16 \quad a:45\)  
(b) \(c:12 \quad b:13 \quad 14 \quad d:16 \quad a:45\)

(c) \(14\)  
\[0 \quad 1\]
\(d:16\)  
\(25\)  
\(a:45\)  
\(f:5 \quad e:9\)  
\(c:12 \quad b:13\)

(d) \(25\)  
\[0 \quad 1\]
\(c:12 \quad b:13\)  
\(14\)  
\(d:16\)  
\(f:5 \quad e:9\)

(e) \(a:45\)

(f) \(100\)
\[0 \quad 1\]
\(a:45\)  
\(25\)  
\(55\)  
\(30\)  
\(c:12 \quad b:13\)  
\(14\)  
\(d:16\)  
\(f:5 \quad e:9\)