Chapter 16: Greedy Algorithms - Introduction

- The greedy technique selects an option that appears best at any step in processing.
- The greedy approach is used in optimization problems.
- Note that ”the option that appears best” is a decision based on local information.
  - It may or may not be part of the global optimal solution.
- For some problems, the greedy approach is guaranteed to find an optimal solution, but not for all problems.
- Algorithms that use this technique are called greedy algorithms.
Chapter 16: Greedy Algorithms - The Activity Selection Problem

• Overview
  – The problem deals with scheduling several processes (activities) that need access to some resource
  – Only one process can access the resource at one time
  – Want to schedule the activities to allow the greatest number of activities to proceed

• Formal description
  – Let \( S = \{a_1, a_2, \ldots, a_n\} \) be a set of activities
  – Each \( a_i \) has a start time \( s_i \) and a finish time \( f_i \), \( 0 \leq s_i \leq f_i < \infty \)
    * \( a_i \) is active over the interval \([s_i, f_i)\)
  – Activities \( a_i \) and \( a_j \) are compatible if their intervals do not overlap; i.e., \( s_i \geq f_j \) or \( s_j \geq f_i \)
  – The problem is to find the largest subset of mutually compatible activities from \( S \)
  – Assume \( f_1 \leq f_2 \leq \ldots \leq f_n \)
  – For 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>2</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>( f_i )</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>13</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

* \( a_1, a_4, \) and \( a_6 \) are mutually compatible, but not optimal
  * \( a_2, a_4, a_5, \) and \( a_7 \) are mutually compatible and optimal

• Optimal substructure of the problem
  – Let \( S_{ij} \) be the set of activities that start after \( a_i \) finishes and finish before \( a_j \) starts
  – Want to find the maximal subset of \( S_{ij} \); call it \( A_{ij} \)
  – \( A_{ij} \) will include some \( a_k \)
    * This divides \( S_{ij} \) into three sets: \( S_{ik}, \{a_k\}, \) and \( S_{kj} \)
    * The problem now becomes one of finding \( A_{ik} \subseteq S_{ik} \) and \( A_{kj} \subseteq S_{kj} \)
      * See text p 416 that proves this generates an optimal solution
    * Then \( A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj} \)
    * \(|A_{ij}| = |A_{ik}| + |A_{kj}| + 1\)
This problem could be solved using dynamic programming
- Let $c[i, j] = |A_{ij}|$
- Then $c[i, j] = c[i, k] + c[k, j] + 1$
- The problem is to find values for $a_k$ that lead to an optimal solution
  
  $$c[i, j] = \begin{cases} 
  0 & |S_{i,j}| = 0 \\
  \max_{a_k \in S_{i,j}} \{c[i, k] + c[k, j] + 1\} & |S_{i,j}| > 0
  \end{cases}$$

- If we solve this using dynamic programming, many of the $c[i, j]$ will be used multiple times in the solution

Greedy approach
- In the greedy approach, will choose what appears to be the best choice at each step
- For this activity problem, the best choice is the one that uses the resource for the least amount of time
- For this discussion, this will be the activity that finishes earliest
- Once an activity has been selected, the problem reduces to finding - from among the activities that are mutually compatible with the choices made to this point - the next activity that finishes earliest
- Since this problem exhibits optimal substructure, we know that if earlier choices are part of the optimal solution, then the optimal solution from what remains will also be part of that optimal solution

**Theorem 16.1:** Consider any non-empty subproblem $S_k$ and let $a_m$ be an activity in $S_k$ with the earliest finish time. Then $a_m$ is included in some maximum-size subset of mutually compatible activities of $S_k$. 

- Proof p 418
- Based on this theorem, we can simply iteratively choose an activity with the earliest finish time from the set of mutually compatible activities and add it to the solution.
- Because the entries are ordered by finish time, each activity will only need to be visited once
Chapter 16: Greedy Algorithms - The Activity Selection Problem (3)

- Recursive algorithm:

  Recursive-Activity-Selector (s, f, k, n)
  {
  1   m = k + 1
  2   while (m <= n AND s[m] < s[k])
  3       m = m + 1
  4   if (m <= n)
  5     return union(a[m], Recursive-Activity-Selector(s, f, m, n))
  6   else
  7     return null
  
  - Initial call is Recursive-Activity-Selector(s, f, 0, n)
  - f[0] is initialized to 0
  - k keeps track of where the algorithm last left off
  - Run time: Θ(n)

- Iterative algorithm:

  Greedy-Activity-Selector (s, f)
  {
  1   n = S.length
  2   A = a[1]
  3   k = 1
  4   for (m = 2 to n)
  5     if (s[m] >= f[k]) {
  6       A = union(A, a[m])
  7       k = m
  8     }
  8   return A
  

Chapter 16: Greedy Algorithms - Greedy Strategy

- Steps used in the Activity Selection Problem:
  1. Determine the optimal substructure of the problem
  2. Develop a dynamic programming style recursive solution
  3. Show that if make a greedy choice, only one subproblem remains
  4. Prove that it is always safe to make a greedy choice (i.e., choice will lead to optimal solution)
  5. Develop a recursive algorithm that implements the greedy strategy
  6. Convert the recursive algorithm to an iterative one

- In the dynamic programming approach used in the Activity Selection Problem, started by examining subproblems
  - Instead, could have considered greedy approach immediately:
    Structure an optimal solution about a single subproblem

- General guidelines for greedy design:
  1. Formulate the problem so that when make a problem-solving choice, only a single subproblem exists
  2. Prove there is always an optimal solution to the original problem that makes a greedy choice, demonstrating that the greedy choice is safe
  3. Demonstrate that the problem has optimal substructure
    - Show that if combine optimal solution to subproblem already solved with greedy choice, have optimal solution to original problem

- Cannot always guarantee that greedy techniques will solve a particular optimization problem
  - But if problem exhibits the greedy choice and optimal substructure properties, good chance that it will
• Greedy choice property
  – Can produce a globally optimal solution by making locally optimal choices
  – This is the main difference WRT dynamic programming solutions
    * DP problems usually solved bottom-up by generating optimal solutions to multiple small subproblems and combining
    * In greedy approach, make the best choice and find an optimal solution to a single subproblem
      • This solution does not depend on future choices or solutions to other subproblems
    * This is an inherently top-down approach
  – Must prove that the greedy choice at each step leads to an optimally global solution
  – Greedy solutions are usually more efficient than DP solutions, often using preprogramming or use of appropriate data structures

• Optimal substructure
  – Proof is usually easier than for DP, since only a single subproblem

• In designing a solution to an optimization problem, must be careful that
  1. Do not create a DP solution when a greedy solution will work
  2. Do not mistakenly create a greedy solution when a DP approach is required
Chapter 16: Greedy Algorithms - Greedy Strategy (3)

• The knapsack problem:
  – Two versions:
    1. 0-1 (all-or-nothing)
    2. Fractional - Can take all or part of an object
  – Both versions exhibit optimal substructure, but only the fractional version can be solved by greedy techniques
  – Consider a problem where max weight is 50, and three objects:

<table>
<thead>
<tr>
<th>item</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>wt</td>
<td>10</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>cost</td>
<td>60</td>
<td>100</td>
<td>120</td>
</tr>
<tr>
<td>density</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

  – If base choices on cost density (cost/weight)
    * In fractional problem, will take all 10 of first, 20 of second, and 20 of third, for total worth of 60 + 100 + 80 = 240 cost
  – In 0-1 approach (based on choosing smallest items first), take all 10 of the first, and all 20 of the second for a total of 160 cost
    * But the optimal solution is all 20 of the second and all 30 of the third for 220 total cost
  – The 0-1 problem is a candidate for DP, where want to consider solutions to other choices, not just the single greedy one
Chapter 16: Greedy Algorithms - Huffman Codes

• A Huffman code is a variable-length binary code
  – Each code is a unique binary string called a codeword
  – Principle is that will use shorter codewords for higher frequency items, and longer codewords for those of lower frequency
  – This approach leads to less memory requirements than for fixed-length codes

• Huffman code is a prefix code:
  No codeword is a prefix for any other codeword
  – This guarantees no ambiguity

• Consider

<table>
<thead>
<tr>
<th>char</th>
<th>a</th>
<th>e</th>
<th>n</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequency</td>
<td>0.18</td>
<td>0.23</td>
<td>0.07</td>
<td>0.52</td>
</tr>
<tr>
<td>code</td>
<td>011</td>
<td>00</td>
<td>010</td>
<td>1</td>
</tr>
</tbody>
</table>

• Can represent code as a binary tree
  – Codeword interpreted as a simple path from root to leaf

```
         1.00
        /   \
       0     1
      /   \   /   \
     0     S  E    1
    / \
   N  A
```
Chapter 16: Greedy Algorithms - Huffman Codes (2)

• Optimal code for a file is always represented by a full binary tree:
  Every internal node has two children
  - If $C$ is a character set, then an optimal binary tree has
    * $|C|$ leaves
    * $|C| - 1$ internal nodes
  - Given such a tree, the number of bits required for a file is calculated as follows:
    * Let $c.freq$ be the frequency of occurrences of character $c \in C$
    * Let $d_T(c)$ be depth of $c$’s leaf (the length of path from root to $c$)
    * Then cost of tree $B$ is
      \[
      B(T) = \sum_{c \in C} c.freq \cdot d_T(c)
      \]

• Algorithm for constructing Huffman codes:

\[
\text{Huffman (C)}
\{
\begin{align*}
1 & \quad n = \text{C.length} \\
2 & \quad Q = \text{C} \quad \text{//Q is min priority queue based on pct/freq} \\
3 & \quad \text{for (i = 1 to } n - 1) \{ \\
4 & \quad \quad z = \text{new node} \\
5 & \quad \quad z.left = x = \text{Extract-Min}(Q) \\
6 & \quad \quad z.right = y = \text{Extract-Min}(Q) \\
7 & \quad \quad z.freq = x.freq + y.freq \\
8 & \quad \quad \text{Insert}(Q, z) \\
9 & \quad \} \\
& \quad \text{return Extract-Min}(Q) \quad \text{//root of tree}
\end{align*}
\]

- Technique:
  * Create a new node that is parent of two cheapest entries from queue
  * Insert node back into queue with value equal to the sum of its children
  * Left child always item of lesser cost
  * Continue until a single node in the queue
Chapter 16: Greedy Algorithms - Huffman Codes (3)

- Analysis:
  * Min heap can be initialized in $O(n)$ time
  * Loop (lines 3 - 8) executes $n-1$ times and each heap operation $\in O(lg n)$ for $O(nlg n)$ run time
  * So overall run time $\in O(nlg n)$
Correctness:

- Lemma 16.2:
  Let $C$ be an alphabet where each $c \in C$ has frequency $c.freq$. Let $x$ and $y$ be characters of lowest frequency. There exists an optimal prefix code for $C$ in which $x$ and $y$ have the same length and differ only in the last bit.

- Proof:
  * Let $a$ and $b$ be two sibling characters at max depth in tree $T$
  * Assume that $a.freq \leq b.freq$ and $x.freq \leq y.freq$
  * Then $x.freq \leq a.freq$ and $y.freq \leq b.freq$
  * Could have $x.freq = a.freq$ or $y.freq = b.freq$, but not $x.freq \leq b.freq$ (in which case all four could be the same)
  * So $x \neq b$
  * Will transform $T \rightarrow T' \rightarrow T''$ by swapping $x$ with $a$, then $y$ with $b$

  * Note that if $x = a$ but $y \neq b$, $x$ and $y$ would not end up at same max depth
  * $B(T) - B(T') = \sum_{c \in C} c.freq * d_T(c) - \sum_{c \in C} c.freq * d'_T(c)$
    
    $= x.freq * d_T(x) + a.freq * d_T(a) - x.freq * d_{T'}(x) - a.freq * d_{T'}(a)$
    
    $= x.freq * d_T(x) + a.freq * d_T(a) - x.freq * d_T(x) - a.freq * d_T(x)$
    
    $= (a.freq - x.freq)(d_T(a) - d_T(x))$

  because $x.freq$ is the min frequency and $d_T(a)$ is the max depth
  * A similar argument can be made for transforming $T$ to $T'$

\[\]
Chapter 16: Greedy Algorithms - Huffman Codes (5)

* So \( B(T) \leq B(T'') \), but since \( T \) is optimal, \( B(T'') = B(T) \) so \( T'' \) is also optimal
* This lemma implies that can build an optimal tree by merging optimal values

- **Lemma 16.3:**
  Let \( C \) be an alphabet where each \( c \in C \) has frequency \( c.freq \). Let \( x \) and \( y \) be characters of lowest frequency. Let \( C' \) be alphabet \( C \) with \( x \) and \( y \) replaced with new character \( z \). So \( C' = C - \{x, y\} \cup \{z\} \). Let \( z.freq = x.freq + y.freq \). Let \( T' \) be any tree representing an optimal prefix code for \( C' \). Then tree \( T \), derived from \( T' \) by replacing leaf \( z \) with an internal node with \( x \) and \( y \) as children, represents an optimal prefix code for \( C \).

- **Proof:**
  * For each \( c \in C - \{x, y\} \), \( d_T(c) = d_{T'}(c) \) implies that \( x.freq \cdot d_T(x) + y.freq \cdot d_T(y) \)
  * Since \( d_T(x) = d_T(y) = d_T(z) + 1 \), then
  * \( x.freq \cdot d_T(x) + y.freq \cdot d_T(y) = (x.freq + y.freq)(d_{T'}(z) + 1) = z.freq \cdot d_{T'}(z) + (x.freq + y.freq) \)
  * Thus \( B(T) = B(T') + x.freq + y.freq \), or \( B(T') = B(T) - x.freq - y.freq \)
  * Suppose \( T \) doesn’t represent optimal code for \( C \)
    - Then there is an optimal tree \( T'' \) where \( B(T'') < B(T) \)
    - Assume \( T'' \) has \( x \) and \( y \) as siblings
    - Let \( T''' = T'' \) with \( z \) replacing the parent of \( x \) and \( y \) and \( z.freq = x.freq + y.freq \)
    - Then \( B(T''') = B(T'') - x.freq - y.freq < B(T) - x.freq - y.freq = B(T') \)
    - But this contradicts the assumption that \( T' \) represents an optimal prefix code
    - Therefore \( T \) must represent an optimal code for \( C \)

- **Theorem 16.4:**
  Procedure \textit{Huffman} produces an optimal prefix code

  - **Proof:** Floows directly from Lemma 16.2 and Lemma 16.3