[Greedy Algorithms]

**Ref:** Chapter 16 of text. (Omit Sections 16.4 and 16.5.)

**Greedy Strategy**

- Used generally in solving optimization problems.
- **Basic idea:** At each step, make a choice that looks best at that step (i.e., a choice that is “locally optimal”).
- Hope that a locally optimal choice will lead to a “globally optimal” solution.
- The idea leads to algorithms for some problems and heuristics for others.

**Notes:**

- An *algorithm* for a problem produces an optimal solution for *every* instance of the problem.
- A *heuristic* for a problem may not produce optimal solutions for some instances.

**Problem 1:** Finding a minimum cost path from source to a destination in a directed graph.

**Greedy strategy:**

- From any node $x$, follow an edge whose cost is the smallest among all the edges leaving $x$.
- If there are two or more edges leaving $x$ with the smallest cost, pick one of the edges arbitrarily (i.e., break ties arbitrarily).

**Note:** Greedy strategy leads only to a heuristic as shown in the following example.

![Graph](image-url)

Greedy path: $\langle v, p, w \rangle$  Cost = 10
Optimal path: $\langle v, q, w \rangle$  Cost = 8
Problem 2: Minimum Set Cover.

Given: A set $X = \{x_1, x_2, \ldots, x_n\}$ and a collection $S = \{s_1, s_2, \ldots, s_m\}$ of subsets of $X$.

Required: A subcollection $S'$ of $S$ containing the smallest number of sets such that the union of the sets in $S' = X$.

Greedy strategy: At any time, pick a set $s_i$ from $S$ that covers the maximum number of new elements of $X$.

Algorithm: (Initially $S'$ is empty.)

repeat

(a) Pick a subset $s_i$ of largest size in $S$. (Break ties arbitrarily.)

(b) Remove $s_i$ from $S$ and add it to $S'$.

(c) From each remaining set in $S$, delete the elements of $s_i$.

until (Union of the sets in $S' = X$).

Note: In general, the greedy algorithm for Minimum Set Cover does not produce an optimal solution.

Example:

Let $X = \{x_1, x_2, x_3, x_4, x_5\}$. Further, let

\[
s_1 = \{x_1, x_2, x_3\} \quad s_2 = \{x_4, x_5\}
\]
\[
s_3 = \{x_1, x_3, x_5\} \quad s_4 = \{x_2, x_5\}
\]
\[
s_5 = \{x_4, x_5\}.
\]

Greedy solution: $\{s_3, s_4, s_5\}$ (uses 3 subsets).

Optimal solution: $\{s_1, s_2\}$ (uses 2 subsets).
Activity Scheduling Problem (ASP):

- For activity (or task) $a_i$, a time interval $[s_i, f_i]$ is specified ($s_i$ : start time, $f_i$ : finish time).
- A pair of activities is compatible if the corresponding time intervals do not overlap (except possibly at the end points).

**Formal definition:** Two activities $a_i$ and $a_j$ with respective intervals $[s_i, f_i]$ and $[s_j, f_j]$ are compatible if and only if $s_i \geq f_j$ or $s_j \geq f_i$.

Example:

```
  a1
    □
      a2
        □
          a3
```

- Compatible tasks: $a_1$ and $a_3$, $a_1$ and $a_4$, $a_2$ and $a_4$, $a_3$ and $a_4$.
- Incompatible tasks: $a_1$ and $a_2$, $a_2$ and $a_3$.

**Statement of ASP:**

**Given:** A set $S = \{a_1, a_2, \ldots, a_n\}$ of $n$ activities; for each activity $a_i$, an interval $[s_i, f_i]$ ($1 \leq i \leq n$).

**Required:** A subset $S'$ of $S$ of maximum cardinality such that every pair of activities in $S'$ is compatible.

**A greedy strategy for ASP:**

At any step, choose a task $a_i$ that satisfies both of the following properties:

1. Task $a_i$ is compatible with the tasks already chosen.
2. Among the tasks satisfying Condition 1, $a_i$ has the earliest finish time (greedy choice).

**Note:** Intuitively, the greedy choice leaves as much opportunity as possible to pick future activities.
**A greedy algorithm for ASP:** Handout 14.1.

Running time:
- Step 1: $O(n \log n)$ time (sorting step).
- Steps 2 and 3: $O(1)$ time.
- Step 4: $O(n)$ time. (The for loop runs $n - 1$ times and the time spent in each iteration is $O(1)$.)
- Step 5: $O(n)$ time (since $|A| \leq n$).
- Overall time: $O(n \log n)$.

**Correctness:** We must prove that
- Every pair of tasks in $A$ is compatible.
- $|A|$ has the maximum possible value.
- Corresponding lemmas stated below. (Proofs are in Handout 14.2.)

**Lemma 1:** Every pair of tasks in the set $A$ returned by the algorithm is compatible.

**Lemma 2** (Greedy Choice Property):
Let $\langle a_1, a_2, \ldots, a_n \rangle$ denote the task ordering after the sorting step. There is an optimal solution to the problem that includes task $a_1$.

**Lemma 3** (Optimal Substructure Property):
If $A$ is an optimal solution to the problem containing $a_1$, then $A - \{a_1\}$ is an optimal solution to the problem consisting of the set $S_2$ of tasks where $S_2 = \{a_i : s_i \geq f_1\}$. (That is, $A - \{a_1\}$ is an optimal solution to the subproblem containing all the tasks that are compatible with $a_1$.)

**Greedy vs Dynamic Programming:**
- Greedy choice generally works in a “top-down” fashion while dynamic programming is a “bottom-up” method.
- Both greedy approach and dynamic programming rely on an optimal substructure property.
0-1 and Fractional Knapsack Problems:

Given: A collection $C$ of $n$ items $\{I_1, I_2, \ldots, I_n\}$; for each item $I_j$, there is a value $v_j$ (dollars) and weight $w_j$ (lbs), $1 \leq j \leq n$; a knapsack that can hold at most $W$ lbs.

0-1 Knapsack Problem: Find a subcollection $C'$ of items such that the total weight of items in $C'$ is at most $W$ and the total value of the items in $C'$ is maximized. (Think of each item as a bar of gold.)

Fractional Knapsack Problem: Find a solution of weight at most $W$ and maximum value, possibly using fractions of items. (Think of each item as a bag of gold dust.)

Notes:
- Both 0-1 Knapsack and Fractional Knapsack exhibit optimal substructure property.
- 0-1 Knapsack is an NP-complete problem. There is a greedy algorithm for the Fractional Knapsack problem.

A greedy algorithm for Fractional Knapsack:

Basic idea: Sort the items in decreasing order of the ratio value/weight and add items to the knapsack in that order.


Correctness: Exercise.

Running time: $O(n \log n)$.
Note: The greedy strategy used for the Fractional Knapsack problem does not work for 0-1 Knapsack problem.

Example: There are three items \((I_1, I_2\text{ and } I_3)\) and \(W = 50\).

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>(I_1)</td>
<td>60</td>
<td>10</td>
</tr>
<tr>
<td>(I_2)</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>(I_3)</td>
<td>120</td>
<td>30</td>
</tr>
</tbody>
</table>

The list in decreasing order of ratios is \((I_1, I_2, I_3)\).

- Greedy solution: \(\{I_1, I_2\}\) Value = 160.
- Optimal solution: \(\{I_2, I_3\}\) Value = 220.

Note: The 0-1 Knapsack problem can be solved using dynamic programming. (The approach is similar to that for the Subset Sum problem.)

Huffman Codes:
- Widely used in data compression.
- Based on variable length prefix codes.

Example: Consider a file containing a string of length 1000 composed of letters a through e.

The frequencies (i.e., the number of occurrences) of the letters are as follows:

- a: 100
- b: 200
- c: 100
- d: 100
- e: 500

A simple encoding scheme:
- With 5 letters, we can use three bits per letter.
- The number of bits used by this scheme to store the string of length 1000 is equal to \(1000 \times 3 = 3000\).
An alternative scheme:

<table>
<thead>
<tr>
<th>Letter</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>100</td>
</tr>
<tr>
<td>b</td>
<td>110</td>
</tr>
<tr>
<td>c</td>
<td>101</td>
</tr>
<tr>
<td>d</td>
<td>111</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
</tr>
</tbody>
</table>

Example: The string “abecd” would be encoded as 1001100101111.

Notes:
- The above coding scheme uses a prefix code; i.e., no codeword is a prefix of another codeword. (This is the property that enables us to decode.)
- Basic idea: Short codewords for most frequently occurring symbols.
- Using this scheme, the string of 1000 characters uses only 2000 bits.

Binary Tree Representation:

Notes:
- The length of a codeword for a symbol is the depth of the corresponding leaf (i.e., the number of edges in the path from the root to the leaf) in the tree $T$.
- The cost of the tree (i.e., the total number of bits needed to encode the string) is denoted by $B(T)$.

Goal: Find a tree $T$ for which $B(T)$ is minimized.
**Formula for** $B(T)$:

$$ B(T) = \sum_{c \in C} d_T(c) f(c) $$

where

- $f(c)$ is the frequency of symbol $c$ and
- $d_T(c)$ is the depth of the leaf corresponding to $c$ in $T$.

**Huffman’s Algorithm (Idea):**

- Let $|C| = n$. Start with a leaf for each symbol $c \in C$. (No. of leaves = $n$.)
- Carry out $n - 1$ ‘merges’ of pairs of nodes to create the final tree.

**Pseudocode:** Handout 14.1.

**Running time:** $O(n \log n)$ using a heap.

**Correctness:** Follows from the lemmas stated below. (Proofs to be discussed in class.)

**Lemma 4:** Every optimal solution to the problem is a full binary tree (i.e., each internal node has exactly two children).

**Lemma 5:** (Greedy Choice Property)

Let $x$ and $y$ be two symbols in $C$ with the two lowest frequencies. There is an optimal prefix code for $C$ in which the codewords for $x$ and $y$ are of the same length and differ only in the last bit.

**Lemma 6:** (Optimal Substructure Property)

Let $T$ be a full binary tree representing an optimal prefix code for $C$. Let $x$ and $y$ be two symbols in $C$ that are sibling leaves in $T$ and let $z$ be their parent. Then, considering $z$ as a symbol with frequency $f(z) = f(x) + f(y)$, the tree $T' = T - \{x, y\}$ represents an optimal prefix code for $C' = (C - \{x, y\}) \cup \{z\}$. 