Greedy Algorithms

Ex. Continuous Knapsack

Before we seek to

1. Maximize \( \sum_{i=1}^{n} x_i v_i \)

2. Subject to \( \sum_{i=1}^{n} x_i w_i \leq W \)

where \( W > 0 \), \( v_i > 0 \), and \( w_i > 0 \) \((1 \leq i \leq n)\). However instead of \( x_i \in \{0,1\} \) we now allow \( 0 \leq x_i \leq 1 \) for \( 1 \leq i \leq n \).

A vector \( x = (x_1, \ldots, x_n) \) will be called a feasible solution if (2) is satisfied without regard to the optimality condition (1).

A greedy strategy consists of making a locally optimal (greedy) choice, then solving the subproblem arising from this choice.

In this problem that means including the "best" object which does not exceed the capacity constraint, then doing the same thing with the remaining objects and remaining capacity.
Knapsack \((v, w, W)\)

1. \(n \leftarrow \text{length}[v]\)
2. \(\text{for } i \leftarrow 1 \text{ to } n\)
3. \(x[i] \leftarrow 0\)
4. \(\text{weight} \leftarrow 0\)
5. \(\text{while weight} < W\)
6. \(i \leftarrow \text{the "best" remaining object}\)
7. \(\text{if weight} + w[i] \leq W\)
8. \(x[i] \leftarrow 1\)
9. \(\text{weight} \leftarrow \text{weight} + w[i]\)
10. \(\text{mark i as included.}\)
11. \(\text{else}\)
12. \(x[i] \leftarrow \frac{W - \text{weight}}{w[i]}\)
13. \(\text{weight} \leftarrow W\)
14. \(\text{return } x\)

The "best" object in 6) can be interpreted in several ways.

In general we define a selection function \(f(i)\) which encodes the desirability of object \(i\). Line 6) then maximize \(f(i)\) over all remaining (unmarked) objects.
Some possible choices for $f$ in this example are:

- $f(i) = v_i$ (process ordered in order of decreasing values in greedy loop.)
- $f(i) = \frac{1}{w_i}$ (increasing weights.)
- $f(i) = \frac{v_i}{w_i}$ (decreasing value to weight ratio.)

Ex. $n=5$, $W=10$.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_i$</td>
<td>2</td>
<td>3</td>
<td>6.6</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>$w_i$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>$v_i/w_i$</td>
<td>2</td>
<td>1.5</td>
<td>2.2</td>
<td>1</td>
<td>1.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$f(i)$</th>
<th>$x$</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_i$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$v_i/w_i$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$v_i/w_i$</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

In this example (at least) $f(i) = \frac{v_i}{w_i}$ gives the best result. Is this optimal?
**Theorem**

If we maximize \( \frac{v_i}{w_i} \) on line (6) then Knapsack returns an optimal solution.

**Proof:**

Without loss of generality we may assume the objects are indexed by decreasing \( \frac{v_i}{w_i} \):

\[
\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \ldots \geq \frac{v_n}{w_n}
\]

Let \( x = (x_1, \ldots, x_n) \) be the solution returned by Knapsack.

If all \( x_i = 1 \), then \( x \) is clearly optimal.

Otherwise let \( j \) be the first index such that \( x_j < 1 \). Inspecting the algorithm it's clear that:

\[
x_i = 1 \quad \text{for } 1 \leq i < j
\]
\[
x_j < 1
\]
\[
x_i = 0 \quad \text{for } j < i \leq n
\]

And that:

\[
\sum_{i=1}^{n} x_i w_i = W
\]
Let \( V(x) = \sum_{i=1}^{n} x_i v_i \), the value of \( x \). Let \( y = (y_1, \ldots, y_n) \) be any other feasible solution, and \( V(y) = \sum_{i=1}^{n} y_i v_i \) its value.

We must show \( V(x) \geq V(y) \), and hence \( x \) is optimal.

Since \( y \) is feasible, \( \sum_{i=1}^{n} y_i w_i \leq W \), and therefore
\[
\sum_{i=1}^{n} (x_i - y_i) w_i = W - \sum_{i=1}^{n} y_i w_i \geq 0.
\]

Now
\[
V(x) - V(y) = \sum_{i=1}^{n} (x_i - y_i) v_i \\
= \sum_{i=1}^{n} (x_i - y_i) w_i \left( \frac{v_i}{w_i} \right),
\]

Observe that

\( i < j \) \( \Rightarrow \) \( x_i = 1 \) \( \Rightarrow \) \( x_i - y_i \geq 0 \) and \( \frac{v_i}{w_i} \geq \frac{v_j}{w_j} \)

\( \therefore (x_i - y_i) \left( \frac{v_i}{w_i} \right) \geq (x_i - y_i) \left( \frac{v_j}{w_j} \right) \)

\( i > j \) \( \Rightarrow \) \( x_i = 0 \) \( \Rightarrow \) \( x_i - y_i \leq 0 \) and \( \frac{v_i}{w_i} \leq \frac{v_j}{w_j} \)

\( \therefore (x_i - y_i) \left( \frac{v_i}{w_i} \right) \geq (x_i - y_i) \left( \frac{v_j}{w_j} \right) \)

\( i = j \) \( \Rightarrow \) \( (x_i - y_i) \left( \frac{v_i}{w_i} \right) = (x_i - y_i) \left( \frac{v_i}{w_i} \right) \)
Thus \( (x_i - y_i) \left( \frac{v_i}{w_i} \right) = (x_i - y_i) \left( \frac{v_i}{w_i} \right) \) for \( 1 \leq i \leq n \).

\[
\begin{align*}
\sqrt{x} - \sqrt{y} & \geq \sum_{i=1}^{n} (x_i - y_i) w_i \left( \frac{v_i}{w_i} \right) \\
& = \left( \frac{v_i}{w_i} \right) \sum_{i=1}^{n} (x_i - y_i) w_i \\
& \geq 0
\end{align*}
\]

\( x = (x_1, \ldots, x_n) \) is optimal, as required. \( \Box \)

Examining lines 2-3 and 5-12, one sees that knapsack runs in time \( \Theta(n) \).

Our dynamic programming solution to the 0-1 knapsack problem ran in time \( \Theta(nW) \), since the table was of size \( n \times (W+1) \).

Despite the 0-1 knapsack problem can be solved more efficiently using a greedy strategy.
Ex. Same as before, but now 0-1 Knapsack. 

\[ n = 5, \ W = 10 \]

\[
\begin{array}{cccccc}
V & 2 & 3 & 6.6 & 4 & 6 \\
W & 1 & 2 & 3 & 4 & 5 \\
\sqrt{W} & 2 & 1.5 & 2.2 & 1 & 1.2 \\
\end{array}
\]

Greedy Solution:
\[ X = (1, 1, 1, 0, 0) \]
\[ \{ \text{Value} = 15.6 \}, \text{Weight} = 10 \]

Exercise:
Check that dynamic program yields the very same solution.

Ex.: \( n = 5, \ W = 11 \)

\[
\begin{array}{cccccc}
V & 1 & 6 & 18 & 22 & 28 \\
W & 1 & 2 & 5 & 6 & 7 \\
\sqrt{W} & 1 & 3 & 3.6 & 3.67 & 4 \\
\end{array}
\]

Greedy Solution:
\[ X = (0, 1, 0, 0, 1) \]
\[ \{ \text{Value} = 35 \}, \text{Weight} = 10 \]

Exercise:
Check that dynamic program yields the solution
\[ Y = (0, 0, 0, 1, 1, 0) \]
\[ \{ \text{Value} = 40 \}, \text{Weight} = 11 \]