Dynamic programming: The Knapsack Problem

The input to the “Knapsack Problem” is a list \([w_1, \ldots, w_n]\) of weights, a list \([v_1, \ldots, v_n]\) of values, and a weight limit \(W\). All these numbers are positive reals.

The idea is that we have a bag ("knapsack") and \(n\) items; item \(i\) has weight \(w_i\) and value \(v_i\). Our job is to put as much value in the bag as we can while not exceeding the weight limit \(W\). Formally, we need to solve the following optimization problem.

\[
\max_{S \subseteq [n]} \left\{ \sum_{k \in S} v_k \left| \sum_{k \in S} w_k \leq W \right. \right\}
\]

where \([n] = \{1, \ldots, n\}\).

Explanation. We seek to select a subset \(S \subseteq \{1, \ldots, n\}\) (set of items to put in the bag).

\[
f(S) := \sum_{k \in S} v_k
\]

is the objective function to be maximized, subject to the constraint

\[
g(S) := \sum_{k \in S} w_k \leq W.
\]

We read Equation (1) as follows: choose the subset \(S \subseteq [n]\) so as to maximize the objective function (2) subject to the constraint (3).

The operations we permit: arithmetic and bookkeeping (addition, subtraction, comparison, copying) on integers in the range \(0 \leq \max(n, W)\) and on reals at unit cost.

**Theorem.** Under the assumption that the weights are integers (but the values are real), one can find the optimum in \(O(nW)\) operations.

The solution illustrates the method of “dynamic programming.” The idea is that rather than attempting to solve the problem directly, we embed the problem in an \((n+1) \times (W+1)\) array of problems, and solve those problems successively. The following definition is the brain of the solution.

For \(0 \leq i \leq n\) and \(0 \leq j \leq W\), let \(m[i, j]\) denote the maximum value of the knapsack problem restricted to \(S \subseteq \{1, \ldots, i\}\), under weight limit \(j\).

So we can only select from the first \(i\) items and the weight limit becomes a variable, \(j\).

The heart of the solution is the following recurrence.

\[
m[i, j] = \max\{m[i-1, j], \quad v_i + m[i-1, j-w_i]\}.
\]

Explanation. If in the optimal solution \(i \not\in S\) then \(m[i, j] = m[i-1, j]\); otherwise we gain value \(v_i\) and have to maximize from the remaining objects under the remaining weight limit \(j-w_i\) (assuming \(j \geq w_i\)). The optimum will be the greater of these two values.

It should also be clear that \(m[i, 0] = m[0, j] = 0\) for all \(i, j \geq 0\). With this initialization, a nested pair of for-loops fills in the array of values \(m[i, j]\).

We describe the algorithm in pseudocode.
Initialize (lines 1–6):
01 for i = 0 to n
02 \text{ \hspace{1em} } m[i, 0] := 0
03 end
04 for j = 1 to W
05 \text{ \hspace{1em} } m[0, j] := 0
06 end

Main loops:
07 for i = 1 to n
08 for j = 1 to W
09 \text{ \hspace{1em} if } j < w_i \text{ \hspace{1em} then } m[i, j] := m[i - 1, j] \text{ \hspace{1em} \text{: item } i \text{ \hspace{1em} cannot be selected :} }
10 \text{ \hspace{1em} else } m[i, j] := \text{ as in equation } \heartsuit \text{ \hspace{1em} \text{: heart of solution :} }
11 \text{ \hspace{1em} end }
12 \text{ \hspace{1em} end }
13 \text{ \hspace{1em} return } m[n, W]

The statement inside the inner loop expresses the value of the next $m[i, j]$ in terms of values already known so the program can be executed.

The required optimum is the value $m[n, W]$.

Analysis
Correctness follows by induction on $i$ from Eq. \heartsuit.
Complexity: Evaluating Eq. \heartsuit requires a constant number of operations per entry, justifying the $O(nW)$ claim.