Algorithms – Instructor: László Babai

Dynamic programming: the knapsack problem

The input of the "Knapsack Problem" is a list $[w_1, \dots, w_n]$ of *weights*, a list $[v_1, \dots, v_n]$ of *values*, and a *weight limit* W. All these numbers are positive reals.

The problem is to find a subset $S \subseteq \{1, \dots, n\}$ such that the following *constraint* is observed:

$$\sum_{k \in S} w_k \leq W. \tag{1}$$

The *objective* is to maximize the total value under this constraint:

$$\max \leftarrow \sum_{k \in S} v_k. \tag{2}$$

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

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 $n \times W$ 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 $S \subseteq \{1, \dots, i\}$, under weight limit j. ♣

The **heart** of the solution is the following recurrence.

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

Explanation: if in the optimal solution $i \notin S$ 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 $m[0, k] = m[k, 0] = 0$ for all $k \geq 0$. With this initialization, a double **for**-loop fills in the array of values $m[i, j]$:

Initialize (lines 1–6):

1 **for** $i = 0$ to n
2     $m[i, 0] := 0$
3 **end**
4 **for** $j = 1$ to W
5     $m[0, j] := 0$
6 **end**

Main loops:

7 **for** $i = 1$ to n
8   **for** $j = 1$ to W
9   **if** $j < w_i$ **then** $m[i, j] := m[i-1, j]$ (* item i cannot be selected *)
10   **else** $m[i, j] :=$ as in equation ♡ (* heart of solution *)
11   **end**
12 **end**
13 **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 $m[n, W]$. Evaluating equation ♡ requires a constant number of operations per entry, justifying the $O(nW)$ claim.