Dynamic programming

Dynamic programming

- is used for *optimisation* problems, where we want to find the ‘best way’ of doing something;
- is a recursive approach that involves breaking a global problem down into more local subproblems;
- assumes *optimal substructure* i.e. that there is a simple way to combine optimal solutions of subproblems to get an optimal global solution;
- avoids the inefficiency that straightforward recursion may suffer from *subproblem overlap* (i.e. when decomposition results in the same subproblems occurring often and being solved many times) by:
- *memoizing* (i.e. storing the solutions of subproblems in a table and then looking them up) and:
- computing the table bottom up rather than top down.

Dynamic programming often produces a polynomial-time algorithm for finding the optimal solution when brute force enumeration of possibilities would be exponential.

Solving the 0-1 knapsack problem

Given: a set S = {s_1, s_2, …, s_n} of n items where each s_i has integer value v_i and integer weight w_i .

Required: to choose a subset O of S such that the total weight of the items chosen does not exceed W and the sum of the values v_i of items in O is maximal.

How can we break the global problem down into subproblems in a way that gives subproblem optimality?

Suppose the optimal solution for S and W is a subset O in which s_k is the highest numbered item.

Then O-{s_k} is an optimal solution for S_{k-1} = {s_1, …, s_{k-1}} and total weight W-w_k. And the *value* of the global solution O is v_k plus the value of the subproblem solution.

Given a target weight w and a set S_k = {s_1, …, s_k} imagine examining all the subsets of S_k whose total weight is £ w. Some of these subsets might have bigger total values than others. Let V[k, w] be the biggest total value of such a subset of S_k. Now we give a recursive definition of V[k, w].

V[k, w] = 0 if either k = 0 or w = 0, otherwise

V[k, w] = if $w_k > w$ then V[k -1, w]
else max{V[k -1, w], v_k + V[k -1, w - w_k]}

The recursive definition of V[k,w]

The recursive definition of V[k, w] says that the value of a solution for stage S_k and target weight w either

- includes item s_k, in which case it is v_k plus a subproblem solution for S_{k-1} and total weight w-w_k, or
- doesn’t include s_k, in which case it is just a subproblem solution for S_{k-1} and the same weight w.

The thief can solve the problem by considering the total set S_n and weight W deriving its solution from that of the subproblems S_{n-1} and weight W-w_n or S_{n-1} and weight W. These subproblems in turn …

At each stage S_k,there are two choices to be compared and the better one made:

1. If the thief picks item s_k then she gets the value v_k and can choose from items s_1, …, s_{k-1} up to the weight limit w - w_k, getting additional value V[k -1, w - w_k], for a total value v_k + V[k -1, w - w_k].

2. If the thief decides not to take item s_k then she can choose from items s_1, …, s_{k-1} up to the weight limit w, and get the value V[k -1, w].

Using a table

Inputs = max weight W, item values (v_1, v_2, …, v_n) and weights (w_1, w_2, …, w_n).

We can solve the problem by recursively computing V[k, w]. But to avoid subproblem overlap dynamic programming uses a bottom-up table —

The values V[k, w] are stored in a table V[0..n, 0..W] whose entries are computed in row-major order (i.e. the first row of V is filled in from left to right, then the second row, and so on). This bottom-up approach ensures that subproblems are solved once only.

At the end of the computation, V[n, W] contains the maximum value the thief can take.

Tracing the choices — the subset of items to take can be recovered from the table V by starting at V[n, W] and tracing where the optimal values came from.

If V[k, w] = V[k -1, w] then item s_k is not part of the solution and we continue tracing with V[k -1, w]. Otherwise item s_k is part of the solution and we continue tracing with V[k -1, w - w_k].

Pseudocode

Dynamic_01_knapsack(n, W, v_1, …, v_n, w_1, …, w_n)

1. for w from 0 to W, set V[0, w] = 0
2. for k from 1 to n
3.   set V[k, 0] = 0
4.   for w from 1 to W
5.   if $w_k > w$
6.   then set V[k, w] = V[k -1, w]
7.   else
8.   if V[k -1, w] > v_k + V[k -1, w - w_k]
9.   then set V[k, w] = V[k -1, w]
10. else set V[k, w] = v_k + V[k -1, w - w_k]

The algorithm describes how the table is filled in.

It doesn't describe how to trace the choices.

Example

Apply the algorithm to solve the 0-1 knapsack problem:

s_1: $v_1 = 60$ $w_1 = 10$

s_2: $v_2 = 100$ $w_2 = 20$

s_3: $v_3 = 120$ $w_3 = 30$

w k	0	10	20	30	40	50
0						
1						
2						
3						