Algorithms. Lecture Notes 4

Dynamic Programming for Subset Sum and the Knapsack Problem

As an indication that dynamic programming (and nothing simpler) will be needed for the Knapsack problem, we begin with a natural greedy algorithm and a small but impressive example of an instance where it miserably fails. Since we have to pack as much value as possible in a limited space, it is tempting to re-index the items such that $v_1/w_1 \geq v_2/w_2 \geq \ldots \geq v_n/w_n$ and take the best items one by one until the knapsack is full. However, consider the following instance, amazingly with only two items: $v_1 = 10\epsilon$, $w_1 = \epsilon$, $v_2 = 90$, $w_2 = 10$, $W = 10$. The optimal solution is item 2 with value 900, but the above greedy algorithm would take item 1 which has a better value-per-weight ratio, and this rules out the profitable item 2. By making $\epsilon > 0$ arbitrarily small, we get arbitrarily bad greedy solutions.

Let us turn to dynamic programming instead. First we consider the Subset Sum problem in the case when an exact sum is required: Given numbers W and w_i, $i = 1, \ldots, n$, find a subset whose sum is exactly W, or confirm that no solution exists. We assume that all these numbers are integer. (Arbitrary rational numbers can be multiplied with their greatest common divisor, without changing the problem.) It is convenient to call W the capacity and to imagine that we pack items of sizes w_i in a knapsack.

The obvious idea for dynamic programming is: Consider the items in the given order and decide whether to choose the current item or not. But, in contrast to Interval Scheduling, it is not enough to use j as the only argument in our recursion formula: Our decisions influence the remaining capacity, and we have to keep track on the capacity as well. Therefore we need a second argument in our "dynamic programming function". We define: $P(j, w) = 1$ if some subset from the first j items has the sum w, and $P(j, w) = 0$ else. Our function has Boolean values 1 (true) and 0 (false).

There is nothing to optimize, we only want to know (a) whether a solution *exists* and (b) in the positive case we want some solution.

To **define** a suitable function is only the first step in the process of designing a dynamic programming algorithm. The second step is to find an efficient rule to actually **compute** the values of this function, for any given instance. The value that we eventually want is $P(n, W)$. Suppose that we have already computed the $P(i, y)$ for all $i < j$ and $y < w$. If we do not choose the jth item, we just copy the solution for $j - 1$. If we choose the jth item, the capacity used up before this step was by w_i units smaller. Since these are the only possible cases, we can compute each $P(j, w)$ by the following Boolean expression:

$$P(j, w) = P(j - 1, w) \vee P(j - 1, w - w_j).$$

Initialization is trivial: $P(0, w) = 0$ for all $w > 0$, and $P(j, 0) = 1$ for all j, since the empty set is a solution with sum 0. We can also assume $P(j - 1, w - w_j) = 0$ for $w < w_j$, because no solution with negative size exists.

Note that the number nW of sub-instances is still reasonably small. In every step we need to know which is the current item, and how much capacity is already used, and this information is enough. For each pair j, w it is completely irrelevant which of the previous items we have taken. Again, this avoids combinatorial explosion.

The "art" of dynamic programming is to recognize such parameters that limit the number of sub-instances to be considered for the given problem. This is the creative step which requires some problem analysis. But once we have found suitable parameters, the development of the algorithm is usually pretty straightforward.

Back to our problem: In the case that $P(n, W) = 1$, we can reconstruct a solution by backtracing. The total time complexity is $O(nW)$, since the computation of every $P(j, w)$ needs $O(1)$ operations. However, be aware that this is not a polynomial time bound! Number W is exponential in its length, which is $O(logW)$ digits. Hence nW cannot be polynomially bounded in the input length. Still, if $W < 2^n$ then the dynamic programming algorithm is faster than exhaustive search. And $W < 2^n$ is often true in practical instances.

Next we consider the more general optimization version of Subset Sum: If no subset has exactly the desired sum W, compute a subset with the largest possible sum below W. ("Pack a knapsack as full as possible.") The

only new twist is that we must memoize the optimal sum rather than a Boolean value. Accordingly, we define $OPT(j, w)$ as the largest number $\leq w$ that can be a sum of values w_i of a subset of the first j items. Without much further explanation it should be clear that:

$$OPT(j, w) = \max\{OPT(j-1, w), OPT(j-1, w-w_j) + w_j\},$$

with initialization $OPT(0, w) = 0$ for all w, and $OPT(j, 0) = 0$ for all j. We can also assume $OPT(j-1, w-w_j) = 0$ for $w < w_j$.

Now we are ready to solve the general Knapsack problem with sizes w_j and profit values v_j, almost as a byproduct of our discussion. Define $OPT(j, w)$ to be the maximum total *value* of a subset from the first j items with total size at most w. Because only some minor modification is necessary, we give the recursive formula straight away:

$$OPT(j, w) = \max\{OPT(j-1, w), OPT(j-1, w-w_j) + v_j\}.$$

Finally, consider a variant of the Knapsack problem where arbitrarily many copies of every item are available. Surprisingly, yet another slight modification of the recursive formula solves it immediately:

$$OPT(j, w) = \max\{OPT(j-1, w), OPT(j, w-w_j) + v_j\}.$$

Why is it correct? We leave it to you to think about it.

Dynamic Programming for Sequence Comparison

The "linear" structure of the Sequence Comparison problem immediately suggests a dynamic programming approach. Naturally, our sub-instances are the pairs of prefixes of A and B, and we try to align them with a minimum number of mismatches. Accordingly, we define $OPT(i, j)$ to be the minimum number of edit steps that transform $a_1 \ldots a_i$ into $b_1 \ldots b_j$. What we want is $OPT(n, m)$. But here, already the construction of a recursion formula for $OPT(i, j)$ requires some effort and problem analysis.

An idea for a complete case distinction is to consider the "fate" of the last character of A. While transforming A into B by edit steps, what can happen to a_n? We consider two cases: a_n is deleted or not. (1) We may delete a_n, and transform $a_1 \ldots a_{n-1}$ into B. (2) We may keep a_n, and convert it into another character or not. In this case, a_n or its "conversion product" is some character b_j of B.

We consider two subcases: $j = m$ or $j < m$. (2.1) If $j = m$, we have to transform a_n into b_m, and $a_1 \dots a_{n-1}$ into $b_1 \dots b_{m-1}$. (2.2) If $j < m$, then the edit sequence must have created b_m. (Since $j < m$, and the order of characters is preserved, no character of A can be turned into b_m.) Since b_m is a new character, we have to transform A into $b_1 \dots b_{m-1}$. Now we have covered all cases.

We define an auxiliary function by $\delta_{ij} = 1$ if $a_i \neq b_j$, and $\delta_{ij} = 0$ if $a_i = b_j$. Now the above case distinction can be readily expressed a recursive formula, if applied to the currently last positions i, j rather than to n, m:

$$OPT(i,j) = \min\{OPT(i-1,j)+1, OPT(i-1,j-1)+\delta_{ij}, OPT(i,j-1)+1\}.$$

Note that, in the middle case, a_i is mapped onto b_j, so we need an edit step if and only if this character changes. Initialization is done by $OPT(i,0) = i$ and $OPT(0,j) = j$. As usual, the time complexity is the array size, here $O(nm)$, and an optimal edit sequence can be recoverded from the stored edit distances $OPT(i,j)$ by backtracing: Starting from $OPT(n,m)$ we review which case gave the minimum, and construct the alignment of A and B from behind. (It is recommended to do some calculation examples.)

Problem: Searching

Given: a set S of n elements, and another element x.

Goal: Find x in S, or report that x is not in S.

Motivations:

Searching is, of course, a fundamental problem, appearing in database operations or inside other algorithms. Often, S is a set of numbers sorted in increasing order, or a set of strings sorted lexicographically, or any set of elements with an order relation defined on it.

Problem: Skyline

Given: n rectangles, having their bottom lines on a fixed horizontal line.

Goal: Output the area covered by all these rectangles (in other words: their union), or just its upper contour.

Motivations:

This is a very basic example of problems appearing in computer graphics. The rectangles are front views of skyscrapers, seen from a distance. They may partially hide each other, because they stand in different streets. We want to describe the skyline.

Such basic graphics computations should be made as fast as possible, as they may be called many times as part of a larger graphics programme, of an animation, etc.