Algorithms. Lecture Notes 4

Dynamic Programming Algorithms for Subset Sum and Knapsack

A new feature of the next examples of dynamic programming is that the "dynamic programming function" has two parameters rather than one. This is quite typical. We will also see that the function is not always numerical; it can also have Boolean values.

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 counterexample 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 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 90, 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 integers. (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 jth item or not. But, in contrast to Interval Scheduling, it is not enough to use j as the only argument in our objective function: Our decisions influence the remaining capacity, thus we must keep track of the capacity as well. Therefore we need a second argument, and 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 here. We only want to know whether some solution *exists*.

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 do choose the jth item, the capacity used up before this step was by w_i units smaller. Since these are the only possible options, it is correct to compute each $P(j, w)$ by the following Boolean expression:

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

As for the initialization, we have $P(0, w) = 0$ for all $w > 0$, and we have $P(j, 0) = 1$ for all j, since the empty set is always 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.

The number nW of sub-instances to consider is 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 making the remaining choices.

The "art" of dynamic programming is to recognize such parameters that limit the number of sub-instances to consider 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 (in the same way as earlier). The total time complexity is $O(nW)$, since the computation of every $P(j, w)$ needs $O(1)$ operations.

However, be aware that $O(nW)$ is not a polynomial time bound! Number W is exponential in its description length, since we need only $O(\log W)$ digits to write W. Hence nW cannot be polynomially bounded in the instance size. Still, if $W < 2^n$ then the dynamic programming algorithm is faster than exhaustive search. The condition $W < 2^n$ is often satisfied in practical instances.

Next we consider a 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 just a Boolean value. Accordingly, we define $OPT(j, w)$ as the largest number not exceeding w that can be obtained as a sum of values w_i of some 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 the initialization $OPT(0, w) = 0$ for all w, and $OPT(j, 0) = 0$ for all indices j. To take care of the case $w - w_j < 0$ we can set $OPT(j, y) = -\infty$ for all j and for all $y < 0$.

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 previous discussion. Define $OPT(j, w)$ to be the largest possible 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.

Problem: Segmentation

This is a generic scheme of problems, rather than one specific problem. Let f be some "easily computable" function that assigns a positive real number to every possible sequence of items. These items can be numbers, characters, or other objects.

Given: a sequence $(x_1, \ldots, x_n)$ of items.

Goal: Partition the sequence into segments $(x_i, \ldots, x_j)$ so that the sum of the values $f((x_i, \ldots, x_j))$ of all these segments is maximized/minimized.

Motivations:
f can be interpreted as a quality measure or a penalty for segments. Our segmentation shall maximize the total quality, or minimize the penalty. We mention a few concrete problem examples:

- *Data analysis:* A sequence of real numbers shall be partitioned into segments that ascend or descend almost linearly. The penalty for every segment is measured by the deviation from the closest linear function (regression line) by, e.g., the sum-of-sqares error. (This problem is treated in Section 6.4 of the textbook.)
- *Parsing:* A text without spaces shall be partitioned into words. The penalty for a segment is, e.g., its edit distance to the most similar real word in a dictionary.

Dynamic Programming for Segmentation Problems

We consider the penalty minimization version first. Let e_{ij} denote the penalty for segment $(x_i \ldots, x_j)$ in the given sequence. Being already trained in dynamic programming, we define $OPT(j)$ to be the smallest possible sum of penalties in a segmentation of $(x_1, \ldots, x_j)$. The last segment may start at any position $i \leq j$, therefore we have:

$$OPT(j) = \min_i OPT(i-1) + e_{ij},$$

where the minimum is taken over all i with $1 \leq i \leq j$. A new phenomenon is that we make a **multi-way choice** in every step. The number of cases is no longer constant, and it takes $O(j)$ time to evaluate every $OPT(j)$. Thus, the time complexity is $O(n^2)$, plus the time for computing

all e_{ij}. It depends on the penalty function how difficult these computations are.

In enhanced versions of segmentation problems, only some maximum number of segments may be allowed in a segmentation. Then, our dynamic programming formula needs a second parameter counting the segments we have already used up.

Problem: Sequence Comparison (String Editing)

Given: two strings $A = a_1 \dots a_n$ and $B = b_1 \dots b_m$, where the a_i, b_j are characters from a fixed, finite alphabet.

Goal: Transform A into B by a minimum number of edit steps. An edit step is to insert or delete a character, or to replace a character with another one.

The *edit distance* of A and B is the minimum number of necessary edit steps. The problem can be reformulated as follows. We define a *gap* symbol that does not already appear in the alphabet. An *alignment* of A and B is a pair of strings A' and B' of equal length, obtained from A and B by inserting gaps before, after or between the symbols. A *mismatch* in an alignment is a pair of different symbols (real symbols or gaps) at the same position in A' and B'. Then, our problem is equivalent to computing an alignment of A and B with a minimum number of mismatches.

Generalized versions of the problem assign costs to the different edit steps. The costs may even depend on the characters.

Motivations:

- *Searching and information retrieval:* Finding approximate occurrences of keywords in texts. Keywords are aligned to substrings of the text. Mismatches can stem from misspellings or from grammatical forms of words.
- *Archiving:* If several, slightly different versions of the same document exist, and all of them shall be stored, it would be a waste of space to store the complete documents as they are. It suffices to store one master copy, and the differences of all versions compared to this master copy. The deviations of any document from the master copy are described in a compact way by a minimum sequence of edit steps.
- *Molecular biology:* Comparison of DNA or protein sequences, searching for variants, computing evolutionary distances, etc.