Algorithms. Lecture Notes 5

An Algorithm for Weighted Interval Scheduling

After the Interval Scheduling success we dare to attack a more general problem: Weighted Interval Scheduling. Let us try and follow the Erliest End First algorithm: Sort the intervals such that $f_1 < f_2 < \ldots < f_n$. Because of the different weights v_i it is no longer true that we can always put the first interval in an optimal solution X. This interval could have a small weight and intersect some later, more profitable intervals. This makes the problem essentially more difficult than Interval Scheduling. But can we extend solutions of smaller instances to larger instances in some other way?

We may decide for each interval in the sequence to add it to X or not. This sounds like exhaustive search. However, a striking observation regarding the "interval structure" of the problem limits this combinatorial explosion: Once we have decided the status the first j intervals and obtained several possible sets of disjoint intervals with the same rightmost f_i ($i \leq j$), it suffices to keep only one of these partial solutions, namely one with maximum total weight. (This is a crucial moment! Make sure that you fully understand why this is correct.) Hence, at any time we have to memoize at most n partial solutions (one for every f_i), rather than some exponential number.

Now we state the resulting algorithm, along with the correctness arguments, in a more formal notation. For $j = 1, \ldots, n$, let $OPT(j)$ denote the maximum weight we can achieve by selecting disjoint intervals from the first j intervals, i.e., from those with endpoints $f_1 < f_2 < \ldots < f_j$. We will inductively compute every $OPT(j)$ from the previously computed $OPT(i)$, $i < j$. Trivially, we have $OPT(1) = v_1$. Now suppose that all $OPT(i)$, $i < j$, are already known. For the jth interval $[s_j, f_j]$ we have two options: to add it to the solution or not. If we don't, then the best total value is, clearly, the maximum of all $OPT(i)$, $i < j$. (There is no reason to consider any partial solution worse than that.) Even simpler: Since $OPT(1) \leq OPT(2) \leq OPT(3) \leq \ldots$, the optimum is $OPT(j-1)$ in this case. If we decide to put $[s_j, f_j]$ in the solution, we can add v_j to the total value, but we have to make sure that the new interval does not intersect an earlier one. For this step we need some auxiliary function: Let $p(j)$ be the largest index i such that $f_i < s_j$. Then we can take the known solution

with value $OPT(p(j))$ and add the new interval. Altogether we have shown that the following formula is correct:

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

This part of the algorithm amounts to a simple for-loop, with all $OPT(j)$ stored in an array. Of course, prior to this calculation we must compute and store all the $p(j)$ in another array. (The v_j, s_j, f_j are already given in arrays.) It is easy to compute the $p(j)$ in a single scan: We also sort the s_j in ascending order. Then we determine, for every j, the largest $f_i < s_j$. Since we have sorted the s_j, it suffices to move a pointer in the sorted array of the f_i. Hence we can compute all $p(j)$ in $O(n)$ time, plus the time for sorting. The for-loop that computes the $OPT(j)$ values needs $O(n)$ time; this should be obvious: In every iteration we do one addition and one comparison. (Here we assume that addition and comparison of two numbers are elementary operations.)

Note that the formula in the for-loop is recursive: $OPT(j)$ is computed by recurring to function values for smaller arguments. But beware: It would be a big practical mistake to implement this formula in a recursive fashion, i.e., as a subroutine with recursive calls to itself! What would happen? Every call creates two new calls, so that the process splits up into a tree of independent calculations, where the same $OPT(j)$ are computed over and over again in many different branches (unless our compiler is optimized in the way that it recognizes repeated calls with the same input parameter and just returns the function value). The time would be exponential, and we abandon the whole idea that made the algorithm efficient, namely that every $OPT(j)$ needs to be computed only once. This example illustrates the importance of *understanding the structure* of an algorithm. It is not enough to hack formulas in the computer.

Now, have we solved our problem? No. We have computed $OPT(n)$, but how do we get a subset of disjoint intervals that realizes this profit? An obvious idea is: Whenever we compute and store a new value $OPT(j)$, we also store a corresponding set of intervals. (We know whether the jth interval has been added or not.) However, this would require many copy operations and add a factor $O(n)$ to our time bound, resulting in $O(n^2)$ time. Compared to exponential time this is still good, however, unnecessarily slow. Surprisingly we can construct a solution much faster, using only the stored values $OPT(j)$: Remember how we obtained $OPT(n)$. We compared two values, and depending on which was larger, we took the nth interval or not. Only by reviewing the OPT values we see which decision had led to the optimum. Next we review either $OPT(j-1)$ or $OPT(p(j))$ in the same way, and we find out whether the considered interval was taken or not. And so on. In other words, we trace back the sequence of optimal decisions. This procedure gives us some optimal solution in another $O(n)$ steps.

Dynamic Programming versus Greedy

The scheme used in the above algorithm is called **dynamic programming**, mainly for historical reasons. It can be characterized as follows.

For a given instance of a problem, we consider *all* solutions of sub-instances that *may* be part of an optimal overall solution. It is enough to keep one optimal solution to every sub-instance. These solutions are extended to larger sub-instances in an incremental fashion. A recursion formula specifies how to compute the optimal value from the already known values for smaller sub-instances.

This approach works well if we can limit the number of sub-instances to consider, ideally by a polynomial bound. (This distinguishes dynamic programming from exhaustive search.) These sub-instances are often defined by some natural restrictions, like the number of items, or some size bound.

An array is filled step by step with the optimal values for the sub-instances. The time complexity is simply the size of this array, multiplied by the time needed to compute each value. Although this array displays only the *values* of optimal solutions, an actual solution is easy to reconstruct in a **backtracing** procedure where we examine on which way the optimum has been reached. The time for backtracing is smaller than the time for computing the optimal values, as we have to trace back only one path in the array.

This outline may still appear a bit nebulous. The best way to fully understand dynamic programming is to study a number of problem examples of different nature, as we will do now. At some point one should notice that the basic scheme is always the same, only the recursion formula and other specific details depend on the problem.

Dynamic programming can be viewed as restricted exhaustive search, but also as an extension of the greedy paradigm. Instead of following only one path of currently optimal decisions, which may or may not lead to an optimal overall solution, we follow all such paths that might bring us to the optimum. Of course, this is feasible only if there are not too many paths to follow. It is very rewarding to learn this technique. Whereas greedy algorithms work only for relatively few problems, dynamic programming has considerably more applications. Our examples are taken from different domains.

Problem: Knapsack

Given: a knapsack of capacity W, and n items, where the ith item has size (or weight) w_i and value v_i.

Goal: Select a subset S of these items that fits in the knapsack (i.e., with $\sum_{i \in S} w_i \leq W$) and has the largest possible sum of values $v = \sum_{i \in S} v_i$.

Motivations:

- Packing goods of high value (or high importance) in a container.
- Allocating bandwith to messages in a network.
- Placing files in fast memory. The values may indicate access frequencies.
- In a simplified model of a consumer, the capacity is a budget, the values are utilities, and the consumer asks himself what he could buy to maximize his happiness.

Problem: Subset Sum

Given: n numbers w_i, $(i = 1, \ldots, n)$ and another number W. (All w_i are positive, and not necessarily distinct.)

Goal: Select a subset S of the given numbers, such that $\sum_{i \in S} w_i$ is as large as possible, but no larger than W. In particular, find out whether there is even a solution with $\sum_{i \in S} w_i = W$.

Motivations:

- This is a special case of the Knapsack problem where $v_i = w_i$ for all i. The goal is to make use of the capacity as good as possible.
- Manufacturing: Suppose that we want to cut up n pieces of lengths w_i $(i = 1, \ldots, n)$, and among our raw materials there is a piece of length W. How can we cut off some of the desired lengths, so that as little as possible of this raw material is left over?
- Political decisions: A committee from several countries makes decisions by weighted majority, where the weight of each country is determined by, e.g., its population size. Can it happen that countries with exactly half of the weight say yes/no?

Dynamic Programming for Subset Sum and Knapsack

A new feature of the next examples of dynamic programming is that we will need two indices rather than one, which is quite typical. We will also see that the recursion formula is not always a numerical function. 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 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 that $P(j - 1, w - w_j) = 0$ 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.