Algorithms. Lecture Notes 3

An Algorithm for Weighted Interval Scheduling

After the Interval Scheduling success we dare to attack a more general problem –Weighted Interval Scheduling. The natural first thought is to try and generalize the successful algorithm (EEF) directly. Sort the intervals such that $f_1 < f_2 < \ldots < f_n$. However, 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. The weights "add another dimension" to the problem. But can we perhaps extend solutions of smaller instances to larger instances in some more sophisticated 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: Suppose that we have decided the status of the first j intervals (to be in X or not). Consider all partial solutions where $[s_j, f_j] \in X$. Among all these (exponentially many!) partial solutions that end in f_j, it suffices to keep only one, namely one with maximum total weight. Why is this correct? Let X_j denote such an optimal partial solution that ends in f_j, and let X be any overall solution that contains $[s_j, f_j]$. If X_j is not a subset of X, then we can replace the subset of intervals in X that end before f_j with X_j, to obtain a solution that is no worse.

In the following we state the resulting algorithm, along with the correctness arguments.

(1) Defining the objective value:

For $j = 1, \ldots, n$, we define $OPT(j)$ to be the maximum weight that can be achieved by selecting a subset of disjoint intervals from the first j intervals, i.e., from those with endpoints $f_1 < f_2 < \ldots < f_j$. (Note carefully that this definition does not require to put $[s_j, f_j]$ in the solution; this interval may be chosen or not.) Our final goal is to evaluate $OPT(n)$.

(2) Computing the objective value:

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 computed. For the 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 $OPT(j-1)$. If we decide to put $[s_j, f_j]$ in the solution, the it contributes v_j to the total value, but we have to make sure that the new interval does not intersect any 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. By the observation above, only this best solution is needed in this case. 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, which 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 above 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 spawns 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. (This does not happen if our *compiler*

recognizes repeated calls with the same input parameter and just returns the function value. But the algorithm itself should not rely on that.) 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 illustrates again 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 the value $OPT(n)$, but how do we get the actual solution, that is, 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 to the solutuon for $OPT(j-1)$ or not.) However, this would require many copy operations and result 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 one 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. By this procedure we can reconstruct 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 certain sub-instances that grow incrementally. For each of these sub-instances it suffices to keep one optimal solution (because, by an exchange argument, no other solution can lead to a better final solution for the entire instance). The optimal values are then computed step by step on the growing sub-instances. A "recursive" formula specifies how to compute the optimal value from the previously computed optimal values for smaller sub-instances. However, it is not applied recursively, rather, the values are stored in an array, and calculations happen in a for-loop.

This approach is efficient 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.

The time complexity is simply the size of the array of optimal values, multiplied by the time needed to compute each value.

Although this array displays only the optimal *values*, 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 no larger 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, and only the recursive 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.

Explaining Dynamic Programming Algorithms

Look again at the algorithm description in the previous paragraph. It consists of twp parts: (1) Defining the objective value. (2) Computing the objective value. Note that these are two different actions!

When you explain an own dynamic programming algorithm, make sure that you write down also part (1), since otherwise it is in general not clear *what* you want to compute in part (2), let alone verification of correctness.

For example, assuming that $p(j)$ is defined, would you understand with ease what "$OPT(j) = \max\{OPT(j-1), OPT(p(j)) + v_j\}$" does, without being told that $OPT(j)$ is supposed to be "the maximum weight that can be achieved by selecting a subset of disjoint intervals from the first j intervals"?

More generally, all newly introduced methematical symbols must be defined before they are used. (Otherwise, how does the poor reader know what they mean?) This should be obvious, but is easy to forget.

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 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?