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. 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.

A new feature of the next examples 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.

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?

Problem: Sequence Comparison (or String Editing)

Given: two strings $A = a_1 \ldots a_n$ and $B = b_1 \ldots b_m$, 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.

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 $f((x_i, \ldots, x_j))$ of all these segments is maximized/minimized.

Motivations:

f is 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 course book.)

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.