Algorithms. Lecture Notes 2

An Algorithm for Interval Scheduling

A naive algorithm would examine all subsets of the given set of intervals and therefore run in $O(n2^n)$ time. Let us try and develop a much, much faster algorithm.

Here is a good heuristic question for algorithm development in general: *Suppose that we are already able to solve the problem for smaller instances, how can we use the partial solutions to solve the overall instance?* Since instances of most computational problems can be split into smaller instances of the same problem in some natural ways, this is a very fruitful approach.

In the case of Interval Scheduling we may ask more specifically: Suppose that we know already how to find the best solution for less than n intervals. Can we perhaps make a decision for one interval (that is, to put it in the solution X) and then solve the remaining instance? However, we have many options to choose this first interval.

A tempting idea is to serve the first request, i.e., the one with smallest s_i. That is, we put this interval $[s_i, f_i]$ in X and, of course, remove all intervals that intersect $[s_i, f_i]$, and continue with the remaining intevals, applying the same rule. But the drawback is obvious: The first interval could be very long. It could even intersect all others.

Perhaps we should take the lengths $f_i - s_i$ into account? Let us make another attempt: First put the shortest interval $[s_i, f_i]$ in X. – Unfortunately, also the shortest interval does not necessarily belong to an optimal solution. The smallest counterexample has only three intervals and is easy to see! So this rule does not work either.

It is time to analyze the reasons for these failures and learn from them. Our selection rules were bad because the selected intervals may overlap too many other intervals. This suggests yet another idea: Put an interval in X that intersects the smallest number of other intervals. This sounds very

plausible, but sadly this rule fails, too, although this time it is a little harder to find a counterexample.

We may try further choices until we are lucky, or we decide to give up at some point. One last attempt: What else could be a good candidate for an interval to put in X?

> *"Ever tried. Ever failed. No matter.*
> *Try again. Fail again. Fail better."*
> (Samuel Beckett)

Reviewing the counterexamples again, we may notice that in the last two attempts the selected intervals were somewhere in the middle of the schedule. What if we come back to the original idea and look at the beginning of the schedule? But taking the interval with earliest starting point was bad. What if we instead take the interval with the earliest endpoint!? The rationale is that this interval is in conflict with the smallest number of other intervals in the remaining instance to the right of its endpoint. For clarity, let us write the proposed algorithm more explicitly:

Earliest End First (EEF): Sort the intervals according to their right endpoints. That is, re-index them so that $f_1 < f_2 < \ldots < f_n$. Put the interval $[s_1, f_1]$ (the one with the smallest f_i) in X, and discard all intervals that intersect this first interval. Repeat this step until every interval is either in X or discarded.

This time we will not detect counterexamples. But after the bad experiences where we saw plausible rules breaking down, it should be clear that we need an **optimality proof**. It is not enough to say that no counterexamples are known. There might exist some, but they might be relatively large and hard to find (as it happened with the last wrong algorithm above). Now, here is a proof:

Assume that there exists a counterexample, that is, an instance where EEF fails to return an optimal solution. Let X be the solution from EEF, and let Y be an optimal solution. By assumption we have $|X| < |Y|$. Let x be the interval with the earliest end. If $x \notin Y$, then we take the leftmost interval $y \in Y$ and exchange it: Let $Y' = Y \setminus \{y\} \cup \{x\}$. Note that Y' is also a set of disjoint intervals (here we need that x has the earliest end), and $|Y'| = |Y|$. Hence Y' is another optimal solution.

This shows that some optimal solution Y' with $x \in Y'$ exists. (Informally, "it is not a mistake" to choose interval x in the first step, as EEF

does.) After removal of x and all intersecting intervals, there remains a smaller instance where EEF will return $X \setminus \{x\}$, whereas $Y' \setminus \{x\}$ is an optimal solution. Since $|X \setminus \{x\}| < |Y' \setminus \{x\}|$, we managed to construct a smaller counterexample!

Hence we have shown: For every counterexample to EEF there exists another counterexample with fewer intervals. On the other hand, some counterexample must have minimal size. This contradiction proves that the initial assumption (existence of some counterexample) was wrong. Thus EEF is correct.

We remark that the same proof can also be formulated as induction on the number of intervals, which is logically equivalent, but the present formulation via a smaller counterexample appears more intuitive and elegant.

The next step is to think about the implementation details that make the algorithm efficient. We may create two copies of each interval, sort them by ascending f_i and ascending s_i, respectively, and put them in two doubly linked lists. The two copies of each interval are connected by pointers, too. Assume again $f_1 < f_2 < \ldots < f_n$. In the f_i list we can always find the smallest end f_1 in $O(1)$ time, as it is simply the first element. The intervals that intersect $[s_1, f_1]$ are exactly those with $s_j < f_1$. That is, we can take the first elements from the other list, until value f_1 is reached. These intervals are deleted in both lists, using the pointers. Since we spend only $O(1)$ time on each copy of an interval, we need $O(n)$ time in total, plus the time for sorting.

To What End Do We Study Algorithm Theory?

Myth: Fast algorithms are not needed (because the hardware is so fast). – Not true!

From comparing the growth of different functions it should be clear that the O-complexity of an algorithm has a larger impact on its practical feasibility than the speed of processors.

Next, since an algorithm must be created only once for a problem but will be applied many, many times, good algorithm design definitely pays off.

So, various computational problems are important, and it is important to solve them by *fast* algorithms. Does that mean that we have to learn a suite of fast algorithms for the most frequent problems? Yes, but this is not enough. Practical problems rarely arise in nice textbook form, and usually

we cannot take an algorithm from the shelves. Often we must adjust or combine algorithms that are known for similar problems or for parts of a given problem. To be able to do this, we need a profound understanding of **how and why** these algorithms work. We have to understand the underlying ideas, not only the particular steps.

Moreover, new computational problems will in general require new algorithms. There is no universal recipe for designing good algorithms, except some general guidelines and techniques. The main part of this course provides some of these general design techniques, a basic toolkit so to speak. We illustrate and practice them on various problem examples. But still the actual algorithm design for a given problem remains a trial-and-error process. (Compare it to craft: Even if one knows one's trade, every application is a bit different.) The selected problems are, hopefully, also of some relevance by their own. But the emphasis is on the design process, rather than on the ready-to-use algorithms for specifc problems. In the Interval Scheduling example we were quite detailed about solution attempts, including failing attempts, in order to stress this creative process and not only the final algorithm.

We also have to learn how to **prove correctness** of a new algorithm.

Myth: Correctness proofs are not needed, it suffices to test algorithms on some instances. – Not true!

Recall the Interval Scheduling example. Various algorithms appeared to be plausibe, but they failed. By not caring about correctness proofs we may happily accept erroneous algorithms. Proofs are not a luxury, just made to intellectually please a few researchers, and testing alone cannot guarantee correctness. We may pick, by good luck, some test instances where our algorithm yields the correct result, but it may have a hidden error that shows up in other instances. This can have fatal consequences, especially in sensible technical systems controlled by algorithms, and so this matter touches even questions of ethics in engineering.

Now we can summarize our main goal: *Develop correct algorithms with low worst-case time bounds, which are, preferably, moderate polynomials.*

About Greedy Algorithms

Earliest End First is an example of a **greedy algorithm**. These are algorithms which, in every step, make the currently best choice, according to some simple optimality criterion. (Once more this is not a formal definition, just a circumscription.) In this sense, greedy algorithms are "myopic".

Myth: Take the best, ignore the rest. If we take an optimal decision in every step, then the overall result will be optimal, too. – Not true!

In fact, *most greedy algorithms are wrong*, and counterexamples are often amazingly small. As we have seen, we do need correctness proofs. The key step of such a correctness proof is often an **exchange argument**: One item of an optimal solution Y is exchanged, in such a way that Y gets closer to the solution produced by the algorithm, without making it worse. For example, in the correctness proof of Earliest End First we took the leftmost interval of Y and replaced it with the interval with the earliest end in the whole instance, which the algorithm would choose.

Problem: Weighted Interval Scheduling

Given: a set of n intervals $[s_i, f_i]$, $i = 1, \ldots, n$, on the real axis. Every interval has also a positive weight v_i.

Goal: Select a subset X of these intervals which are pairwise disjoint and have maximum total weight.

Motivations:

Similar to Interval Scheduling, but here the requests have different importance. The weights might be profits, e.g., fees obtained from the customers.