Algorithms. Lecture Notes 2

An Algorithm for Interval Scheduling

According to the spirit of the course, this section illustrates a possible *process* of algorithm *development* for a problem, rather than giving a good algorithm right away.

A naive algorithm would examine all subsets of the given set of intervals and therefore run in $O(n^2 2^n)$ time, where the n^2 factor accounts for checking the validity of a chosen subset. (Do not worry about the details of a poor algorithm. The inportant fact here is that more than 2^n steps are used.) 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 x (that is, to put it in the solution X) and then solve the remaining instance?

We can express our wish more explicitly: We would like to find some general rule that determines one interval x that we can safely put in X. Then we could remove all intervals that intersect x (this is enforced by the problem), and continue applying our rule until the instance is empty.

However, we have many options to choose this interval x.

A tempting idea is to serve the first request, i.e., let x be the interval with smallest s_i. But the drawback is obvious: This first interval could be very long. It could even intersect all others, and then it is a bad choice.

Perhaps we should take the lengths $f_i - s_i$ into account? Let us make another attempt: Let x be the shortest interval. The intuition is that, typically, short intervals should not intersect many others. – But unfortunately,

"typical" is not enough. The shortest interval does not necessarily belong to an optimal solution either. The smallest counterexample has only three intervals and is easy to see!

It is time to analyze the reasons for these failures and to learn from them. Our selection rules were bad because the selected intervals may overlap too many other intervals. This suggests yet another idea: Let x be some interval 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. The idea of a counterexample is to work with identical copies of some intervals. Since intervals in X must be pairwise disjoint, this does not affect the optimal solution. But by chosing suitable numbers of copies, one can give some interval $x \notin X$ in the middle the smallest number of overlaps, such that the algorithm would wrongly choose x. This way one can construct a counterexample with 11 intervals.

So the third attempt failed, too. We may try further rules until we are lucky, or we may decide to give up at some point. One last attempt: What else could be a good candidate for the interval x?

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

Counterexamples are not unfavorable. While they prove incorrectness of some specific algorithm, they can also give valuable hints on what exactly goes wrong. Reviewing the above cases 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 down the proposed algorithm explicitly:

Earliest End First (EEF): Sort the intervals according to their right endpoints. That is, re-index them such 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 delete all intervals that intersect this first interval. Repeat this step until every interval is either in X or deleted.

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 a **correctness proof**. It is not enough to say that no counterexamples are known. There might exist some, but they might be relatively large and non-obvious (as it happened with the last wrong algorithm above). Now, here is a proof of optimality:

Assume that there exists a counterexample, that is, an instance where EEF fails to return an optimal solution Y. That is, the solution X from EEF has size $|X| < |Y|$. 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 $Y' = Y \setminus \{y\} \cup \{x\}$. Since x has the earliest end, Y' is also a set of disjoint intervals. Moreover, $|Y'| = |Y|$. Hence Y' is another optimal solution. This shows that some optimal solution Y' with $x \in Y'$ exists.

Informally, we have shown that "it is not a mistake" to choose interval x in the first step, as EEF does. It is also worth noticing that the defining property of x is essentially used in the proof. Of course, the rule of the algorithm must play some role.

But so far we have exchanged only one interval. How does this imply correctness of EEF? We can do a proof by contradiction:

Remember that we assumed a counterexample, where EEF returns some set X being smaller than an optimal solution Y. After deletion of x and of all intersecting intervals, there remains a smaller instance where EEF returns $X \setminus \{x\}$. But $Y' \setminus \{x\}$ is a valid solution, too, and $|X \setminus \{x\}| < |Y' \setminus \{x\}|$. Hence the EEF solution is not optimal on this smaller instance either. That is, we have found 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 the minimum 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 might appear more intuitive and elegant.

The next step is to think about the implementation details that make the algorithm efficient. We had already sorted the intervals in such a way that $f_1 < f_2 < \ldots < f_n$. Now we may scan this sorted list from left to right, and record the currently last interval $[s_j, f_j] \in X$. When we read the next f_i, we simply check whether $s_i < f_j$. If so, then we skip $[s_i, f_i]$, since this

interval cannot be in X. If not, then we add $[s_i, f_i]$ to X, according to the rule of EEF. Hence this is our new $[s_j, f_j]$. Since we spend only $O(1)$ time on each interval, we need $O(n)$ time in total, plus the time for sorting.

Note that, in this implementation, intervals not chosen in X are merely skipped rather than "physically" deleted. But this does not make a difference, because skipped intervals are never considered again. The formulation with deletions was better suited for understanding the algorithm itself and its correctness, but for the sake of speed, the proposed implementation handles this detail differently. This illustrates again that one should first care about solving the problem, and only later about programming details and fine-tuning.

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 practicality than the speed of processors.

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

Therefore it is important to solve various computational problems by *fast* algorithms. Does this 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 simply 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. In order to be able to do all 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, and one must extemporize.) 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.

We also have to practice **proving correctness** of new algorithms.

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 results, 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 more precisely: *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!

"Greed is good. Greed is right. Greed works." claimed the broker Gordon Gekko, a character in the 1987 film "Wall Street".

Actually *most greedy algorithms are plainly 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 EEF 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 have chosen). Then, the exchange argument is embedded in an inductive proof or in a proof by contradiction.

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.