Algorithms. Lecture Notes 8

Reductions between Problems

In general, any new problem requires a new algorithm. But often we can solve a problem X using a known algorithm for a related problem Y. That is, we can **reduce** X to Y in the following informal sense: A given instance x of X is translated to a suitable instance y of Y, so that we can use the available algorithm for Y. Eventually the result of this computation on y is translated back, so that we get the desired result for x. This sounds very abstract? Here we illustrate the idea, over a cup of tea:

> *"Problem Y: How does a mathematician prepare tea? Answer: He applies an algorithm: Pour water in the boiler, heat up, take tea leaves, cup and sieve, ... (We skip the remaining details. You certainly know the algorithm.)*
>
> *Problem X: How does the mathematician prepare tea if there is already water in the boiler? Answer: He would pour out the water. This way he has reduced problem X to problem Y."*

Now we give a more serious (but still a bit artificial) example of a reduction. Suppose we want an algorithm for multiplying two integers, and there is already an efficient algorithm available that can compute the square of an integer. It needs $S(n)$ time for an integer of n digits. Can we somehow use it to multiply arbitrary integers a, b efficiently, without developing a multiplication algorithm? Certainly squaring and multiplication are related problems. More precisely, we can use the identity $ab = ((a+b)^2-(a-b)^2)/4$. We only have to add and subtract the factors in $O(n)$ time, apply our squaring algorithm in $S(n)$ time, and divide the result by 4, which can be easily done in $O(n)$ time, since the divisor is a constant. Thus we have reduced multiplication (problem X) to squaring (problem Y) as follows. We have taken an instance of X (factors a, b), transformed it quickly into some instances of Y (namely $a + b$ and $a - b$), solved these instances of Y by the

given squaring algorithm, and finally applied another fast manipulation on the results (addition, division by 4) to get the solution ab to the instance of problem X.

It is essential that not only a fast algorithm for Y is available, but the transformations are fast as well. Note that the time for our multiplication algorithm is $O(S(n))$. The $O(n)$ time transformations are already counted in this time bound, as $S(n)$ is certainly not faster than $O(n)$.

Doing multiplication through an algorithm specialized to squaring is, of course, somewhat strange. But still we get an interesting insight from this reduction: One might conjecture that squares can be computed faster than products of arbitrary numbers, since this problem is only a very special case of multiplication. But due to our reduction, these hopes come to nothing, and we can give firmly a negative answer: Any faster algorithm for squaring would immediately yield a faster algorithm for (general) multiplication. Thus squaring is not easier than multiplication.

Now we have identified two different purposes of reductions: (1) Solving a given problem X with help of an already existing algorithm for a different problem Y. (2) Showing that a problem Y is at least as difficult as another problem X.

Note that (1) is of immediate practical value, and even usual business: Ready-to-use implementations of standard algorithms exist in software packages and algorithm libraries. One can just apply them as black boxes, using their interfaces only, without caring about their internal details. This is nothing but a reduction! Point (2) gives us a way to classify problems by their algorithmic complexity. We can compare the difficulty of two problems without knowing their "absolute" time complexity. If Y is at least as difficult as X, then research on improved algorithms should first concentrate on problem X.

Reductions – Now More Formally

After this informal introduction we approach the abstract definitions of reductions that are needed to build up a **complexity theory** of computational problems.

Let X, Y be any two problems. By $|x|$ we denote the length of an instance x of problem X. We say that X *is reducible to* Y *in* $t(n)$ *time*, if we can do the following in $t(n)$ time for any given x with $|x| = n$: Transform x into an instance $y = f(x)$ of problem Y, and transform the solution of y back into

a solution for x.

Symbol f merely denotes the function describing how an instance is transformed. f must be computable in $t(n)$ time. Note that the time needed by the algorithm for problem Y is not counted in $t(n)$. Only the transformations of instances and solutions are charged. These are the extra costs for using the algorithm for Y, so to speak. Assuming that we have an algorithm for problem Y with time bound $u(n)$, we can solve an instance x of problem X in time $t(|x|) + u(|f(x)|)$.

Loosely speaking we can conclude: If Y is an easy problem and the reduction is fast, then X is an easy problem, too. Conversely, if X is a hard problem, and we have a fast reduction to problem Y, then Y is a hard problem, too.

Have you tried to solve the fundamental graph problems Clique, Independent Set, and Vertex Cover algorithmically? Most probably you have encountered difficulties. We defer the question of how difficult they actually are, but we can already *compare* their complexities by means of reductions.

These comparisons become much easier to handle formally when we restrict attention to so-called **decision problems**. A decision probem is simply a problem that takes an input and has to output a Yes or No answer. (The instance has a solution or not.) This is not a severe restriction. Every optimization problem can be viewed as a series of decision problems. Instead of asking "give me a solution where the objective value is minimized" we can ask "does there exist a solution with objective value at most t?", for various thresholds t. Informally, if the optimization problem is easy to solve, then the corresponding decision problem is also easy, for every threshold t. (We just compare an optimal solution to the threshold.) By contraposition, if already the decision problem is hard, then the corresponding optimization problem is also hard.

Now we define a special type of reductions for decision problems X, Y: We say that X is reducible to Y in $t(n)$ time, if we can compute in $t(n)$ time, for any given x with $|x| = n$, an instance $y = f(x)$ of Y such that the answer to x is Yes *if and only if* the answer to y is Yes. (Loosely speaking, instances x, y of the decision problems X, Y are equivalent.) If the time $t(n)$ needed for the reduction is bounded by a polynomial in n, we say that X is **polynomial-time reducible** to Y.

We illustrate the definition with some simple reductions between the mentioned graph problems, but reformulated as decision problems. Let $G = (V, E)$ be an undirected graph. The Clique problem takes as input a graph G and an integer k and asks whether G contains a clique of at least k nodes.

The Independent Set problem takes as input a graph G and an integer k and asks whether G contains an independent set of at least k nodes.

Inuitively it feels that Clique and Independent Set are only different formulations of the same problem. To make this precise, we show that Clique and Independent Set are polynomial-time reducible to each other. A reduction function is established by $f(G, k) := (\bar{G}, k)$, where $\bar{G}$ is the complement graph of G, that is, the graph obtained by replacing all edges with non-edges and vice versa. Regarding the formalities, note that f has to transform an instance of a problem into an instance of the other problem, and an instance consists here of a graph G *and* an integer k. The transformation is obviously manageable in polynomial time.

The Vertex Cover problem takes as input a graph G and an integer k and asks whether G contains a vertex cover of at most k nodes. We show that Independent Set and Vertex Cover are polynomial-time reducible to each other. The key observation is that $C \subseteq V$ is a vertex cover if and only if $V \setminus C$ is an independent set. Vertex covers and independent sets are complementary sets in the same graph. Hence, a vertex cover of size at most k exists if and only if an independent set of size at least $n - k$ exists (and similarly in the other direction). This gives us a possible reduction: $f(G, k) := (G, n - k)$. This time we did not change the graph. The only work of the reduction function is to replace the threshold k with $n - k$.

These very simple reductions show that all three problems are essentially the same. In particular, they are equally hard.

If a problem X is merely a special case of problem Y, we immediately have a reduction polynomial-time reduction from X to Y. To give an example, Interval Scheduling is a special case of Independent Set, which is seen as follows: Given a set of intervals (requests), we construct a graph with the given intervals as nodes, where two nodes are adjacent whenever the represented intervals are intersecting. We call it the **interval graph** of the given set of intervals. The decision version of Interval Scheduling is: Given a set of intervals and an integer k, does there exist a subset of at least k disjoint intervals? The above graph construction is, in fact, a polynomial-time reduction from Interval Scheduling to Independent Set. The function f describing this reduction transforms the set of intervals into its interval graph, while k is unchanged.

Complexity Classes and Hardness

Comparison by polynomial-time reducibility induces a partial ordering on the class of decision problems, with respect to their complexities: This relation is **transitive**, that is, if X is polynomial-time reducible to Y, and Y is polynomial-time reducible to Z, then X is polynomial-time reducible to Z. This is almost obvious, but we must be a bit careful with the time bounds. To prove transitivity, let f and g be the functions transforming the instances from X to Y and from Y to Z, respectively. Let f and g be computable in time p and q, respectively. By assumption, p and q are polynomials. In order to solve an instance x of X (of size n) with help of an algorithm for Z, we can compute instance $g(f(x))$ of Z and then run the available algorithm. The time needed for the reduction is $p(n) + q(p(n))$. Note that we can bound the input length $|f(x)|$ in the second term only by $p(n)$, since the transformation algorithm that computes $f(x)$ can use $p(n)$ time, and it may use this time to generate such a long instance. However, since p, q are polynomials, $q(p(n))$ is still polynomial in n, hence the entire reduction from X to Z is polynomial.

The "bottom" of the mentioned partial ordering of problems by complexity is the class of "easy" problems. We pointed out earlier that efficient algorithms should need at most polynomial time. Accordingly, we define the **complexity class** $\mathcal{P}$ as the class of decision problems that admit an algorithm which solves every instance x (of size n) correctly and in $O(p(n))$ time, where p is some polynomial. Note that p may depend on the problem, but not on n.

If a given problem is quickly reducible to an easy problem, then the given problem is easy, too. Formally, if a decision problem X is polynomial-time reducible to a decision problem $Y \in \mathcal{P}$, then $X \in \mathcal{P}$. The proof is similar to the transitivity proof. Let p be the polynomial time bound for computing the function f which reduces X to Y, and let t be the polynomial time bound of the algorithm for problem Y. (Now we have to count in the time used by this target algorithm.) Given an instance x of X, with $|x| = n$, we compute $f(x)$ and solve instance $f(x)$ by the algorithm for Y. Now the time bound is $p(n) + t(p(n))$, and this is polynomial in n.

The contraposition says: If X is polynomial-time reducible to Y, and X is *not* in $\mathcal{P}$, we can conclude that Y is not in $\mathcal{P}$ either! Thus, reductions allow us to *prove* hardness of many problems, once we know some hard problem to start with. But can we actually prove that some particular problem is not in $\mathcal{P}$? At least, many natural problems are suspected to be hard in this

sense. No polynomial-time algorithms are known for them. Many graph problems are of this type, and also the Knapsack problem. (Remember that our dynamic programming algorithm for Knapsack was not polynomial in the input length!) They seem to resist all our techniques to create fast algorithms. One cannot see how a correct solution to an instance can be built up from solutions to smaller instances in an efficient way. You are welcome to try, but you will always get stuck at some point. Maybe the methods we have learned are too weak for these problems, or too much ingenuity is needed to find the right way of applying the techniques. The question is: Are we not smart enough, or are the problems intrinsically hard, i.e., outside the class $\mathcal{P}$?

Before we outline the current knowledge around this question, we should reflect upon this activity: Having a good algorithm for some problem is clearly of practical value (provided that the problem is relevant), but why is it meaningful to prove that some problem is hard? Well, such negative results have practical value, too: We would *know* that searching for a fast and exact algorithm is hopeless, and we would concentrate our efforts on fruitful workarounds, like refined heuristics, algorithms for special cases, approximation algorithms, etc. (Compare it to mechanics: For fundamental reasons, a *perpetuum mobile* cannot exist, and this knowledge keeps us from wasting our time with hopeless attempts to build one. Instead, we aim at realistic goals in engineering.) Besides finding the limits of efficient computation, there even exist direct applications of hardness: Some methods in cryptography rely on the fact that certain computational problems are hard to solve, and these are used to build encryption schemes that cannot be broken in reasonable time.

The Complexity Class $\mathcal{NP}$

Almost all "natural" algorithmic problems belong to a certain class of decision problems that includes $\mathcal{P}$ but is apparently larger. Below we introduce this larger complexity class.

It is common to our problems that we can easily **verify** (confirm, certify) already *given* solutions. For example, consider the decision version of Knapsack: Given n items, their weights and values, a capacity W, and a desired total value t, the question is whether some subset of items with total weight no larger than W has a total value of at least t. If somebody supplies us with a solution, we can easily check in polynomial time whether this is

in fact a solution: We simply have to add and compare some numbers. Or consider the Independent Set problem: Given a graph and a number k, we can check in polynomial time whether a given subset I of nodes is a valid solution: Count the nodes in I, compare their number to k, and verify for all nodes $u, v \in I$ that u, v are not joined by an edge. For virtually every natural decision problem we can similarly check an already given solution in a short time. (Consider further problem examples on your own, then you will see.)

The complexity class $\mathcal{NP}$ is defined as the class of decision problems which admit an algorithm that *verifies* a given solution in polynomial time. To be precise, the condition is that we can verify the *existence* of a solution in polynomial time, but we can neglect this subtle difference here.

Some comments on this definition are in order. The verification algorithms are not supposed to *solve* the problems, at least, not in polynomial time. The definition does not say how the solutions are obtained (exhaustive search, a good guess, etc.). It is only concerned with the *verification* of already available solutions. The abbreviatiom $\mathcal{NP}$ stands for **nondeterministic polynomial**, which refers to the interpretation that we may have guessed a solution.

We have $\mathcal{P} \subseteq \mathcal{NP}$. Namely, if we can even *solve* a problem correctly in polynomial time then, trivially, we can also verify in polynomial time that it *has* a solution.

As said above, almost every natural, relevant computational problem belongs to $\mathcal{NP}$, and we have that $\mathcal{P} \subseteq \mathcal{NP}$. Is this inclusion strict?! It would be nice to know $\mathcal{P} = \mathcal{NP}$, since this would mean that all these problems are solvable in polynomial time. Unfortunately, the question is open. Moreover, this is perhaps the most famous open question in Computer Science. Nevertheless we can shed some light on this so-called $\mathcal{P}$-$\mathcal{NP}$ question and classify certain problems as "hard". This is the main subject of the next lecture.