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, such that we can use the available algorithm for Y. Eventually the result of this computation on y is translated back, such that we get the desired result for x. Does this sound too abstract? Here we illustrate the idea by an example:

Suppose that we want an algorithm for multiplying two integers, and there is already an efficient algorithm available that can compute the square of one integer. It needs $S(n)$ time for an integer of n digits. Can we use it somehow to multiply arbitrary integers a, b efficiently, without developing a multiplication algorithm? These assumptions are a bit made up, but we will see below that the example is meaningful. We do not have to specify the function S, but it is clear that $S(n) \geq n$. (Why?)

Certainly, squaring and multiplication are closely related problems. In fact, 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 some problem X (multiplication) to some problem Y (squaring). Namely, we have taken an instance of X (the factors a and b), transformed it quickly into two instances of Y (with operand $a+b$ and $a-b$, respectively), solved these instances of Y by the given squaring algorithm, and finally applied another fast computation to the results (an addition, and a division by 4) to obtain the solution ab to the instance of problem X.

It is crucial that not only a fast algorithm for Y is available, but the transformations are fast as well. Note that the total time for our multiplication algorithm is $O(S(n))$. The $O(n)$-time transformations are already

counted in this time bound.

Doing multiplication through an algorithm specialized to squaring may appear somewhat strange. But 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. In fact, in applications with a lot of squarings (simulations of physical systems?) it would be nice to have such a faster algorithm. But due to our reduction, these hopes come to nothing, and we can firmly give a negative answer: Any faster algorithm for squaring would immediately yield a faster algorithm for (general) multiplication as well. We conclude that squaring is not easier than multiplication!

We have identified two different purposes of reductions: (1) solving a problem X with help of an already existing algorithm for a different problem Y, and (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, by using their interfaces, and without caring about their internal details. Formally this is nothing but a reduction! Point (2) might appear less practical, but it gives us a way to compare the difficulty of problems without determining their "absolute" time complexities (which is often impossible to figure out). It can be useful to know such comparisons, If Y is at least as difficult as X, then research on improved algorithms should first concentrate on the easier problem X. Some applications (as in cryptography) even rely on the hardness of certain computational problems, rather than efficient solvability.

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 instance 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 of x.

Symbol f merely denotes the function describing how an instance is

transformed. It 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, because these are the extra costs for using the algorithm for Y, so to speak. According to the very idea of reductions, the solution algorithm for Y is not part of the reduction. In other words, a reduction is merely an "affair bwtween two problems".

Assuming that we have an algorithm for problem Y with time bound $u(n)$, we can now solve an instance x of problem X in time $t(|x|)+u(|f(x)|)$. Since $|f(x)| \leq t(n)$ (why?), this is bounded by $t(n) + u(t(n))$.

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. In this sense, a reduction allows a comparison of the difficulty of two problems.

These comparisons become much easier to handle formally when we restrict attention to so-called **decision problems**. A decision probem is simply a computational problem that has to output a Yes or No answer (e.g., 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, as follows. 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.

Finally we define reductions between decision problems X and 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. (In other words, 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.

Problem: Clique

A **clique** in a graph $G = (V, E)$ is a subset $K \subseteq V$ of nodes such that all possible edges in K exist, i.e., there is an edge between any two nodes in K.

Given: an undirected graph G.

Goal: Find a clique of maximum size in G.

Motivations:

This is a fundamental optimization problem in graphs. Many other problems can be rephrased as a Clique problem. A setting where it appears directly is the following: The graph models an interaction network (persons in a social network, proteins in a living cell, etc.), where an edge means some close relation between two "nodes". We may wish to identify big groups of pairwise interacting "nodes", because such groups may have an important role in the network.

Problem: Independent Set

An **independent set** in a graph $G = (V, E)$ is a subset $I \subseteq V$ of nodes such that no edges in I exist.

Given: an undirected graph G.

Goal: Find an independent set of maximum size in G.

Motivations:

The same general remarks as for the Clique problem apply. A setting where it appears directly is the following: The graph models conflicts between items, and we wish to select as many as possible items conflict-free. For example: Goods shall be packed in a box, but for security reasons certain goods must not be packed together. How many items can we put in the same box?

Problem: Vertex Cover

A **vertex cover** in a graph $G = (V, E)$ is a subset $C \subseteq V$ of nodes such that every edge of G has at least one of its two nodes in C.

Given: an undirected graph G.

Goal: Find a vertex cover of minimum size in G.

Motivations:

Vertex covers are of interest in "facility location" problems. A toy example is the question: How can we place a minimum number of guards in a museum building so that they can watch all corridors?

Another application field is combinatorial inference. As a bioinformatics example, consider some genetic disease that appears if some rare bad variant of a certain gene is present. Geneticists want to figure out what the bad gene variants are. Their number is expected to be small, as a result of a few unfortunate mutations. Every person carries two copies of the gene. Given the genetic data of a group of persons having the disease, we know that each person has at least one bad variant in his/her pair of genes. Now we can try and explain the data by a minimum number of different bad gene variants.

Reductions Between Some Graph Problems

We illustrate the definition of polynomial-time reducibility by some simple reductions between the mentioned graph problems, 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.

It is rather obvious 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. In other words, vertex covers and independent sets are complement 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 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, we construct a graph with the given intervals as nodes, where two nodes are adjacent whenever the represented intervals intersect. 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 pairwise disjoint intervals? Now it should be clear that 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 remains unchanged.

Note that we cannot reduce Independent Set to Interval Scheduling in the same way. The catch is: Starting from an arbitrary graph G, we cannot always find a set of intervals whose interval graph is exactly G. This is because "most" graphs are not interval graphs. But maybe there exists some non-obvious and tricky reduction nevertheless? Later we will be able to rule out this possibility, too.

Complexity Classes and Hardness

Comparisons by polynomial-time reducibility define 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 not surprising and almost obvious, but we must be a little 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, where p and q are polynomials. In

order to solve an instance x of X (of size n) with the 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 a polynomial in n, hence the entire reduction from X to Z is polynomial. Also note that the instances x and $g(f(x))$ are in fact equivalent.

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

If a given problem X is quickly reducible to an easy problem, then problem X 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 an algorithm for problem Y. (Now we have to count in the time used by this target algorithm.) Given an instance x of X, with size $|x| = n$, 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.

Interestingly, 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. We have no clue how a correct solution to an instance could 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 $\mathcal{P}$? In the following we give a cautious "negative" answer.

The Notion of $\mathcal{NP}$-Completeness

Almost all "natural" algorithmic decision problems belong to a certain class of 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) solutions that are already *given*. For example, consider the decision version of Knapsack: Given n items, their weights and values, a capacity W, and a desired total value k, the question is whether some subset of items with total weight no larger than W has a total value of at least k. 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 pairs of 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.

The complexity class $\mathcal{NP}$ is defined as the class of all decision problems which admit an algorithm that can *verify* every Yes-instance in polynomial time, provided that some "advice" is given, in addition to the input. This "advice" is usually just a solution to the problem instance. In this (typical) case we can say more simply that the mentioned algorithm must verify a given solution in polynomial time.

Some comments on this definition are in order. The verification algorithm is not supposed to *solve* the problem, at least, not in polynomial time. Moreover, the definition does not say *how* the solution is obtained (exhaustive search, a good guess, etc.). It is only concerned with the *verification* of an already available solution. 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 an instance *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". Recall that reductions can be used to compare the difficulty of problems. Now we arrive at the central definition:

A decision problem $Y \in \mathcal{NP}$ is said to be **$\mathcal{NP}$-complete** if every (!) problem $X \in \mathcal{NP}$ is polynomial-time reducible to Y. Informally speaking, $\mathcal{NP}$-complete problems are the hardest problems in $\mathcal{NP}$. We can characterize their hardness as follows:

No $\mathcal{NP}$-complete problem belongs to $\mathcal{P}$, unless $\mathcal{P} = \mathcal{NP}$. Assume for contradiction that some $Y \in \mathcal{P}$ is $\mathcal{NP}$-complete. Then, by definition, all $X \in \mathcal{NP}$ are polynomial-time reducible to Y. But since $Y \in \mathcal{P}$, this implies $X \in \mathcal{P}$ for all $X \in \mathcal{NP}$.

To summarize what we have shown: It is open whether $\mathcal{P} = \mathcal{NP}$ or not, but if not, then no polynomial-time algorithm can exist for $\mathcal{NP}$-complete problems. Since until now nobody could find a fast algorithm for any such problem despite decades of intensive research, it is generally believed that $\mathcal{P} \neq \mathcal{NP}$, and hence all $\mathcal{NP}$-complete problems are really hard.

$\mathcal{NP}$-completeness of any specific problems can be proved via reductions from other such problems, due to the following theorem: If problem Y is $\mathcal{NP}$-complete and polynomial-time reducible to $Z \in \mathcal{NP}$, then Z is also $\mathcal{NP}$-complete. This follows immediately from the definition and from the transitivity of polynomial-time reducibility.

Problem: Satisfiability (SAT)

A **Boolean variable** has two possible values: True (1) or False (0). A **literal** is either a Boolean variable x or its negation $\neg x$. A **Boolean formula** is composed of literals joined by operations AND (conjunction, $\wedge$), OR (disjunction, $\vee$), and perhaps further negations. An **assignment** gives a truth value to every variable in a Boolean formula. An assignment is said to be **satisfying** if the formula evaluates to 1. A **clause** is a set of literals joined by OR. Note that "if-then" conditions can be rewritten as clauses. A Boolean formula is in **conjunctive normal form (CNF)** if it consists of clauses joined by AND. We remark that every Boolean function can be

written equivalently as a CNF formula.

Given: a Boolean formula, either in general form or in CNF.

Goal: Find a satisfying assignment (if there exists some).

Motivations: This is a fundamental problem in logic and related fields like Artificial Intelligence. The following is only an example of a more concrete application scenario.

Certain objects can be described as vectors of Boolean variables, where each variable indicates whether the object has a certain property or not. Suppose that the properties of many objects are stored in a database. Now we want to retrieve an object that satisfies a given set of conditions, expressed as clauses. Before we process an expensive database query, it may be good to check whether the conditions are satisfiable at all, because the given specification may be overconstrained. Furthermore, if the result is positive, we may search the database for occurrences of the satisfying assignments, which is much faster and simpler than testing the conditions for each database entry. (However, the speed depends on the number of satisfying assignments we have to try.)

Appendix

NP Quiz: True or False? (And Why?)

The remarks in the Appendix of Lecture Notes 1 apply also here.

- Consider the following problem. Given three integers x, y, k, decide whether the k-th digit of the product xy, in binary representation, equals 1. Claim: "This problem is in $\mathcal{P}$."
- "The problem in the previous question is in $\mathcal{NP}$."
- "If an optimization problem is in $\mathcal{NP}$ (more precisely: if the corresponding decision problem is in $\mathcal{NP}$), then we can verify in polynomial time that a given optimal solution is, in fact, optimal."