Algorithms. Lecture Notes 8

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. 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? 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.

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 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 their complexities 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** $\mathcal{P}$ 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, n denotes the size of x. 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. 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 partial answer.

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

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) 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 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 *verifies* a given solution in polynomial time.

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". Recall that reductions can be used to compare the difficulty of problems. Now comes 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.)