Algorithms. Lecture Notes 9

The "First" $\mathcal{NP}$-Complete Problem

Still we have not seen any single NP-complete problem which could be a starting point for reductions. For one problem, $\mathcal{NP}$-completeness must be proved directly by recurring to the definition. Historically, the first $\mathcal{NP}$-complete problems came from logic: the Satisfiability (SAT) problem for logical formulae (or circuits). The difficulty of SAT, even when restricted to CNF formulae, can be intuitively explained as follows: We may set any variable to 1 in order to satisfy some clause, but the same variable may appear in negated form in other clauses, and then we cannot use it anymore to satisfy these other clauses. Any decisions on the truth value of some variable in a clause restrict the possibilities of satisfying other clauses. This may end up in conflicts where some clause is no longer satisfiable. Then we have to try other combinations of values, etc. In fact, nobody knows how to solve SAT in polynomial time.

Clearly, SAT belongs to $\mathcal{NP}$: If someone gives us a satisfying truth assignment, we can confirm in linear time (in the size of the formula) that it really satisfies the formula. But SAT is probably not in $\mathcal{P}$. A theorem due to S. Cook says that SAT is $\mathcal{NP}$-complete, even for CNF as input.

The proof is long and very technical, but one can give the rough idea in a few sentences: We have to show that any decision problem $X \in \mathcal{NP}$ is polynomial-time reducible to SAT. Since $X \in \mathcal{NP}$, there exists a polynomial-time algorithm that checks the validity of a given solution to an instance x of X. Like every algorithm, it can run on a machine that performs only extremely simple steps, so simple that the internal state of the machine at any time (contents of memory cells etc.) can be described by Boolean variables. A Boolean formula in CNF, of size polynomial in $|x|$, describes the steps of the computation. This CNF is built in such a way that a satisfying truth assignment corresponds to the successful verification of a solution to x. Hence, the whole construction reduces X to SAT in polynomial time. – We emphasize that this was only a brief sketch of the proof.

Don't worry if you find it cryptic. We only need the statement of Cook's theorem, but not its proof.

A side remark: Without Cook's theorem it would not even be clear that $\mathcal{NP}$-complete problems Y exist at all! Remember that the definition requires that *all* problems of the class are reducible to Y, which is a strong demand. This is also the reason for introducing the "strange" class $\mathcal{NP}$: The considered problems must be limited somehow, but to some class that is still large enough to capture most of the relevant computational problems.

3SAT is as Hard as SAT

Surprisingly, some further restriction does not take away the hardness of the SAT problem: A kCNF is a CNF with at most k literals in every clause. kSAT is the SAT problem for kCNF formulae. We show that 3SAT is still $\mathcal{NP}$-complete, by a polynomial-time reduction from the (more general!) SAT problem for arbitrary CNF.

This reduction is best described by an iterative algorithm doing the transformation. Given a CNF, we do the following as long as some clause C with more than 3 literals exists: We split the set of literals of C in two shorter clauses A and B, append a fresh variable u to A and $\bar{u}$ to B. A fresh variable means that u must not occur in any other clause. It is easy to see that $(A \vee u) \wedge (B \vee \bar{u})$ is satisfiable if and only if $C = A \vee B$ is satisfiable. Similarly, the entire formula is satisfiable before the modification if and only if it is satisfiable after the modification. Hence we have got an equivalent instance of the problem. After a polynomial number of steps we are down to a 3CNF.

Stop! It is not completely obvious that the number of steps is bounded by a polynomial. One needs an argument for that. Here is one possibility. We have not epecified exactly how the literals of C are divided in two graoups. We can always put exactly 2 literals in A and the rest in B. Then $(A \vee u)$ has exactly 3 literals, and $(B \vee \bar{u})$ is strictly shorter than $A \vee B$. Hence we have strictly decreased the total length of the long clauses with more than 3 literals. This can happen only linearly many times.

We can also get rid of clauses with 1 or 2 literals in a polynomial number of steps. Namely, ee can make a clause A artificially longer, similarly as above: Take a fresh variable u and replace A with $(A \vee u) \wedge (A \vee \bar{u})$, this time without splitting the clause. It follows that the version of 3SAT with *exactly* 3 literals per clause remains $\mathcal{NP}$-complete.

Next, what about 2SAT? The above reduction cannot produce an equivalent 2CNF. (Do you see why not?) Actually, 2SAT is in $\mathcal{P}$. It can even be solved in linear time, through a rather nontrivial graph algorithm that we cannot show here.

Some Further $\mathcal{NP}$-Complete Problems

3SAT is an excellent starting point for further NP-completeness proofs. The limitation to three literals per clause makes it nice to handle. Next we reduce 3SAT to Independent Set, thus proving in one go the NP-completeness of Vertex Cover, Independent Set, and Clique.

Let us be given an instance of 3SAT, more precisely, a 3CNF with n variables and m clauses, and with exactly 3 literals in every clause. The reduction constructs a graph as follows.
(1) For each variable we create a pair of nodes (for the negated and unnegated variable), joined by an edge.
(2) For each clause we create a triangle, with the 3 literals as nodes.
These pairs and triangles have together $2n + 3m$ nodes.
(3) An edge is also inserted between any node in a pair (1) and any node in a triangle (2) which are labeled with identical literals.

One can show that the problem instances are equivalent: The 3CNF formula is satisfiable if and only if this graph has an independent set of $k = m + n$ nodes. This needs a little thinking, but the proof steps are straightforward. (For a better understanding it can be advisable to take a little example of a 3CNF, draw the resulting graph, and verify the claimed equivalence for the example and then in general.)

Reductions from a problem X to a probelm Y like this are called "gadget constructions" in the literature, because the building blocks of an instance of X are encoded by "gadgets", which are the building blocks of instances of Y. In our case, variables and clauses of a 3CNF are encoded by node pairs and triangles, which are our gadgets. You are not expected to find such gadget constructions yourself, but only to understand given ones.

The $\mathcal{NP}$-completeness of many problems has been established by chains of such reductions, among them the famous **Traveling Salesman** problem, **Coloring** the nodes of a graph with 3 colors, several partitioning, packing and covering problems, numerical problems like **Subset Sum** and (hence) **Knapsack**, various scheduling problems, and many others. $\mathcal{NP}$-complete problems appear in all branches of combinatorics and optimization. We give another natural example that is also useful for further reductions:

Problem: Set Packing

Given: a family of subsets of a finite set.

Goal: Select as many as possible of the given subsets that are pairwise disjoint.

This resembles Interval Scheduling, but now the given subsets can be any sets, rather than being intervals in an ordered set.

We show that Set Packing is NP-complete, by a polynomial-time reduction from Independent Set: Given a graph $G = (V, E)$ and a number k, we construct the following family of subsets $S(v) \subset E$, one for every node $v \in V$: We define $S(v)$ to be the set of all edges incident with v, the "star with center v", so to speak. Note that two stars are disjoint if and only if their centers are not adjacent. Thus, G contains an independent set of k nodes if and only if we can select k pairwise disjoint sets from this family.

Exponential Time Hypothesis (ETH)

The ETH claims, roughly speaking, that no algorithm can solve 3-SAT in a time better than $O(a^n)$, where $a > 1$ is some constant. (At least, this is a variant of ETH that is easy to formulate. There are other variants of different strengths.)

Like the $\mathcal{P} \neq \mathcal{NP}$ hypothesis, it is not known whether ETH is true, however it is widely accepted as a hypothesis. ETH would obviously imply $\mathcal{P} \neq \mathcal{NP}$, but the converse is not clear.

ETH has the advantage that it claims some explicit lower time bound. Conditional on ETH one can prove lower bounds also for other problems via reductions. However one must be more careful with the time bounds of these reductions.

Some Frequent Misconceptions

To prove $\mathcal{NP}$-completeness of a problem Y, one must give a polynomial-time reduction **from** a known $\mathcal{NP}$-complete problem X to Y, not a reduction from Y **to** X. Remember that Y is harder than X (more precisely: at least as hard) and not easier.

An explanation why the direction of reductions is often confused might be a misconception around the word “reduction”. In every-day use, to “reduce” something usually means to make it smaller, and this may be misunderstood as “making the complexity smaller”, but here it is the other way round! The word “transformation” would perhaps avoid this misunderstanding, but “reduction” is the established term.

Furthermore notice that polynomial-time reducibility is not a symmetric relation. If X is reducible to Y, this does in general not imply that Y is also reducible to X. A reduction goes in only one direction, but *inside* a reduction one must show equivalence of the instances, which involves two directions: (1) If x is Yes then $f(x)$ is Yes, and (2) if $f(x)$ is Yes then x is Yes. Moreover, this must hold true for every instance x of X, whereas not every instance y of Y is required to be some $f(x)$. In other words, function f is not necessarily surjective. All these aspects are easy to confuse, but this is only a matter of carefully learning the definitions and reflecting why they are as they are.

Sometimes it is claimed in reports that $\mathcal{NP}$ means “not polynomial”, which is complete nonsense. Finally, one should carefully distinshuish between “$\mathcal{NP}$-problems” (that is, problems in $\mathcal{NP}$, which also includes $\mathcal{P}$), and “$\mathcal{NP}$-complete problems”. Here, sloppy naming can easily produce wrong statements.

Similarly, sometimes it is claimed that $\mathcal{P} \neq \mathcal{NP}$ would imply that $\mathcal{NP}$-complete problems can only be solved in exponential time. However, such exponential lower bounds are not known.

$\mathcal{NP}$-Completeness; Wrap-Up

What should you (at least) have learned about $\mathcal{NP}$-completeness in a basic algorithms course? You should:

- have understood the concepts on a technical level (not only vaguely that $\mathcal{NP}$-complete problems are “somehow difficult”),
- have understood their relevance,
- be able to carry out *simple* reductions (doing complicated reductions is clearly something for specialized scientists in the field),
- know some representative $\mathcal{NP}$-complete problems,
- know where to find more material.

If, in practice, a computational problem is encountered that apparently does not admit a fast algorithm, it is a good idea to look up existing lists of $\mathcal{NP}$-complete problems. Maybe the decision version Y of the problem at hand is close enough to some problem X in a list, and a polynomial-time reduction from X to Y can be established. Then it is clear that Y must be treated with heuristics, with suboptimal but fast approximation algorithms, or with exact but slow algorithms.

A classic reference with hundreds of $\mathcal{NP}$-complete problems is:
Garey, Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* Freeman, San Francisco 1979.

Other repositories are on the Web.

Appendix

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

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

- "Earlier we have seen dynamic programming algorithms for Subset Sum and Knapsack, but now $\mathcal{NP}$-completeness is claimed. This is a contradiction. Something must be wrong with this theory."
- "The Clique problem is polynomial-time reducible to the Knapsack problem."
- "The Knapsack problem is polynomial-time reducible to the Clique problem."
- "The Independent Set problem is $\mathcal{NP}$-complete for arbitrary graphs. Interval graphs are special graphs. It follows that the Independent Set problem for interval graphs is $\mathcal{NP}$-complete."
- "If, for some problem, we know some powerful heuristic that solves typical instances efficiently, we can conclude that this problem is not $\mathcal{NP}$-complete, because those problems are difficult to solve."