Algorithms. Lecture Notes 9

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

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 grandfather (or grandmother?) of all $\mathcal{NP}$-complete problems comes from logic: the Satisfiability (SAT) problem for logical formulae (or circuits). The difficulty of SAT, even when restricted to CNF formulae, may be explained as follows: If we set a variable to 1 in order to satisfy some clause, it 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 restict the possibilities to satisfy 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 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 stress 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.)

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. The reduction is best described by an iterative algorithm for the transformation. Given a CNF, we do the following as long as a clause C with more than 3 literals exist: We split the set of literals of C in two shorter clauses A, B, append a fresh variable u to A and $\bar{u}$ to B. That is, 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. Hence we have got an equivalent instance of the problem. After a polynomial number of steps we are down to a 3CNF. (The hardest part is to see that the number of steps is really bounded by a polynomial. Note that the new variables blow up the formula. One needs a good argument, for example: The sum of cubes of lengths of all clauses decreases strictly.)

Note that this reduction cannot produce an equivalent 2CNF. 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.

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, that is, a 3CNF with n variables and m clauses. 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 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 some thinking, but the proof steps are straightforward.)

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

The $\mathcal{NP}$-completeness of many problems has been established by chains of such reductions, among them the famous Traveling Salesman problem, several partioning, packing and covering problems, numerical problems like Subset Sum and (hence) Knapsack, various scheduling problems, and much more. $\mathcal{NP}$-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 idependent set of k nodes if and only if we can select k pairwise disjoint sets from this family.

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

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

- have understood the definitions on a technical level, not only informally as a vague idea,
- 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 classical 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, for example:
Crescenzi, Kann. *A compendium of NP optimization problems.*
`http://www.nada.kth.se/~viggo/problemlist/`